به نام آن که جان را حکمت آموخت

پیمان رسولی

داده‌ساختارها و الگوریتم‌ها

Data Structures and Algorithms

محمد قدسی

دانشکده‌ی مهندسی کامپیوتر
دانشگاه صنعتی شریف

seyed hamed hoseinian
Software Engineering pnub

saeeditpnu.blogfa.com

فهرست

۱ روش‌های تحلیل الگوریتم‌ها ۱

۱.۱ زمان اجرای الگوریتم‌ها ۱

۲.۱ به‌ترین و بدترین حالت و حالت متوسط ۳

۳.۱ مرتبه‌ی الگوریتم‌ها ۴

۴.۱ تابع‌های رشد ۵

۵.۱ روش‌های تحلیل الگوریتم‌ها ۹
۱.۵.۱ الگوریتم‌های ترتیبی ۹

۶.۱ الگوریتم‌های بازگشتی ۱۰
۱.۶.۱ مسئله‌ی برج‌های هانوی ۱۱

۷.۱ روش‌های حل رابطه‌های بازگشتی ۱۱
۱.۷.۱ حدس و استقرا ۱۱
۲.۷.۱ تکرار با جای‌گذاری ۱۶
۳.۷.۱ قضیه‌ی اصلی ۱۸
۴.۷.۱ رابطه‌های بازگشتی همگن ۱۹
۵.۷.۱ رابطه‌ها بازگشتی غیر همگن با ضرایب ثابت: ۲۲
۶.۷.۱ حل رابطه‌ها بازگشتی غیر همگن: ۲۳

۲ مرتب‌سازی و مرتبه‌ی آماری ۲۷

۱.۲ الگوریتم‌های مرتب‌ساز ۲۷
۱.۱.۲ درخت تصمیم ۲۸

seyed hamed hoseinian
Software Engineering pnub

saeeditpnu.blogfa.com

۲.۲ الگوریتم‌های مرتب‌ساز خطی ۳۱
۱.۲.۲ الگوریتم Count Sort ۳۱
۲.۲.۲ الگوریتم Radix Sort ۳۲
۳.۲.۲ الگوریتم Bucket Sort ۳۳

۳.۲ الگوریتم‌های مرتب‌ساز مبتنی بر مقایسه ۳۴
۱.۳.۲ الگوریتم quicksort ۳۴
۲.۳.۲ گونه‌ی تصادفی quicksort ۳۷

۴.۲ الگوریتم HeapSort ۴۰
۱.۴.۲ ویژگی‌های یک heap ۴۰

۵.۲ میانه‌ها و مرتبه‌های آماری ۴۴
۱.۵.۲ کمینه و بیشینه ۴۵
۲.۵.۲ انتخاب در زمان متوسط خطی ۴۶
۳.۵.۲ انتخاب در بدترین حالت زمانی خطی ۴۸

۶.۲ مرتب‌سازی خارجی ۵۱
۱.۶.۲ مرتب‌سازی خارجی مبتنی بر ادغام ۵۲
۲.۶.۲ مرتب‌سازی خارجی Polyphase ۵۴

۳ داده‌ساختارها ۵۷

۱.۳ دسته‌بندی ۵۷

۲.۳ داده‌ساختارها برای فرهنگ داده‌ای ۵۸
۱.۲.۳ درخت دودویی جست‌وجو ۵۸
۲.۲.۳ متوسط ارتفاع درخت دودویی جست‌وجو ۶۱

۳.۳ صف اولویت (Priority Queue) ۶۳

۴.۳ تبدیل الگوریتم‌های بازگشتی به غیربازگشتی ۶۵

۵.۳ برنامه‌ی غیربازگشتی ۶۸
۱.۵.۳ حذف آخرین بازگشت (Tail Recursion) ۷۰

۶.۳ درخت قرمز– سیاه (Red-Black Tree) ۷۲

seyed hamed hoseinian
Software Engineering pnub

saeeditpnu.blogfa.com

۱.۶.۳ خواص درخت قرمز– سیاه: ۷۲
۲.۶.۳ دوران ۷۳
۳.۶.۳ درج ۷۵
۴.۶.۳ حذف ۷۸

۴ روش‌های طراحی الگوریتم‌ها ۸۳

۱.۴ استقرای ریاضی ۸۳
۱.۱.۴ خطاهای معمول در اثبات با استقرا ۹۵

۲.۴ طراحی الگوریتم با استقرا ۹۷
۱.۲.۴ کُد گری ۹۷
۲.۲.۴ رابطه‌ی مستقل از حلقه برای اثبات درستی الگوریتم ۱۰۰
۳.۲.۴ محاسبه‌ی مقدار یک چندجمله‌ای ۱۰۲
۴.۲.۴ زیرگراف القایی بیشینه ۱۰۲
۵.۲.۴ نگاشت یک‌به‌یک ۱۰۳
۶.۲.۴ مسئله‌ی «ستاره‌ی مشهور» ۱۰۴

۳.۴ روش تقسیم‌وحل برای طراحی الگوریتم‌ها ۱۰۶
۱.۳.۴ فرش‌کردن صفحه‌ی شطرنجی ۱۰۶
۲.۳.۴ زمان‌بندی دوره‌ی بازی‌ها ۱۰۶
۳.۳.۴ مسئله‌ی برج‌ها ۱۱۰
۴.۳.۴ ضرب دو چندجمله‌ای ۱۱۲
۵.۳.۴ شبکه‌های مرتب‌ساز ۱۱۳

۴.۴ روش برنامه‌ریزی پویا ۱۲۱
۱.۴.۴ ترکیب m از n ۱۲۱
۲.۴.۴ ضرب ماتریس‌ها ۱۲۴
۳.۴.۴ مثلث بندی بهینه‌ی یک چندضلعی محدب ۱۲۷
۴.۴.۴ بزرگ‌ترین زیردنباله‌ی مشترک ۱۳۱
۵.۴.۴ درخت دودویی جست‌وجوی بهینه (Optimal BST) ۱۳۴
۶.۴.۴ مسئله‌ی کوله پشتی (Knapsack Problem) ۱۴۰

۵.۴ روش حریصانه درطراحی الگوریتم‌ها ۱۴۸
۱.۵.۴ ویژگی‌های انتخاب حریصانه ۱۴۹
۲.۵.۴ انتخاب فعالیت‌ها ۱۴۹
۳.۵.۴ مسئله‌های کوله‌پشتی ۱۵۰
۴.۵.۴ مسائل زمان‌بندی ۱۵۱

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۵.۵.۴ الگوریتم هافمن ۱۵۴
۶.۵.۴ الگوریتم حریصانه‌ی تقریبی برای مسئله‌ی بسته‌بندی ۱۵۷
۷.۵.۴ تمرین‌ها ۱۵۷

۶.۴ روش‌های جست‌وجو ۱۶۱
۱.۶.۴ روش پس‌گرد ۱۶۱
۲.۶.۴ درخت بازی ۱۶۲
۳.۶.۴ محدود کردن فضای جست‌وجو ۱۶۴

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

© کلیه‌ی حقوق این جزوه متعلق به دکتر محمد قدسی است و تکثیر و توزیع آن فقط با اجازه‌ی مؤلف مجاز است.

با تشکر فراوان از دانش‌جویان زیر که متن اولیه‌ی این جزوه را از کلاس‌های درس «روش‌های حل مسئله» و «ساختمان‌های داده‌ای و الگوریتم‌ها» در نیم‌سال‌های اول ۷۵–۷۴ و دوم ۷۷–۷۶ تهیه، تایپ و با حروف‌چینی کردند: طلا تفضلی، هشام فیلی، علی‌رضا ملک‌زاده، جلال بنابی بروجنی، محمدرضا صلواتی‌پور، محمد مهدیان، افسانه فضلی، ابوالفضل هادی اسفیگره، حجت قادری، مسعود اسدپور، سیدعلی اکرمی‌فر، آرش رستگار، محمودرضا صانعی‌پور، روزبه پورنادر، آرش رجایان، مسلم کاظمی، حبیب ریشمی، ناصر عزتی، اختای ایلعمی، مهران مهر، سولماز کلاهی، آزاده شاکری، شیوا نجاتی، ساسان دشتی‌نژاد و محمدرضا قهرمانی. از رهاب میرزگنی هم به خاطر رسم تعدادی از شکل‌های این جزوه تشکر می‌شود. این جزوه تماماً با فارسی‌تک حروف‌چینی شده است. از گروه پروژه‌ی فارسی‌تک که این نرم‌افزار را رایگان در اختیار عموم قرار داده‌اند نیز تشکر می‌نماییم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

فصل ۱

روش‌های تحلیل الگوریتم‌ها

هدف‌های تحلیل الگوریتم‌ها به شرح زیر است:

- بررسی رفتار الگوریتم قبل از پیاده‌سازی، از نظر زمان اجرا و مقدار حافظه‌ی مصرفی
- مقایسه‌ی الگوریتم‌ها از نظر کارایی

۱.۱ زمان اجرای الگوریتم‌ها

عوامل زیر در زمان اجرای یک برنامه موثرند:

۱. سرعت سخت افزار،

۲. نوع کامپایلر،

۳. اندازه‌ی داده‌ی ورودی مسئله،

۴. ترکیب داده‌ای ورودی،

۵. پیچیدگی الگوریتم، و

۶. پارامترهای دیگر که تاثیر ثابت در زمان اجرا دارند.

از این عوامل، سرعت سخت افزار و نوع کامپایلر به‌صورت ثابت در زمان اجرای برنامه‌ها موثر هستند. پارامترهای مهم، پیچیدگی الگوریتم است که تابعی از اندازه‌ی مسئله می باشد. ترکیب داده‌ی ورودی را نیز می‌توان با محاسبه‌ی پیچیدگی الگوریتم در بدترین حالت ورودی یا در حالت متوسط در نظر گرفت.

برای بررسی یک الگوریتم تابعی به نام $T(n)$ در نظر می‌گیریم که در آن n اندازه‌ی ورودی مسئله است. مسئله ممکن است شامل چند داده‌ی ورودی باشد. به‌عنوان مثال، اگر ورودی یک گراف باشد، علاوه بر تعداد رأس‌ها (n)، تعداد یال‌های آن گراف (m) هم یکی از مشخصه‌های داده‌ی ورودی است. در این صورت زمان اجرای الگوریتم $T(n,m)$ خواهد بود. ممکن است مسئله بیش از چند پارامتر داشته باشد ولی برای تحلیل مسئله ناچاریم آن‌ها که

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مهم هستند در نظر بگیریم (عمل انتزاع[1].) نکته‌ی دیگر آن است که در تحلیل الگوریتم‌ها، مقادیر ثابت قابل چشم‌پوشی هستند. آن‌چه اهمیت دارد، سرعت رشد زمان اجرای برنامه نسبت به اندازه‌ی ورودی است.

روند تحلیل الگوریتم‌ها را با یک مثال شروع می‌کنیم.

مثال: مرتب‌سازی مبتنی بر درج INSERTION-SORT

```
Insertion_Sort(A, n):
   Input:  A (the array to sort)
           n (size of array)
1  for k := 2 to n
2     do key := A[k]
3        i := k - 1
4        while i > 0 and A[i] > key
5           do A[i+1] := A[i]
6              i := i - 1
7        A[i+1] := key
```

این الگوریتم در واقع طوری عمل می‌کند که در ابتدای مرحله‌ی kام عناصر موجود در درایه‌های ۱ تا $k - 1$ ارایه نسبت به‌هم مرتب هستند و در انتهای آن A[k] در مکان مناسب بین این عناصر درج می‌شود. نحوه‌ی درج به‌این صورت است که A[k] با تعویض با عنصر قبلی‌اش به سمت ابتدای آرایه حرکت می‌کند تا بالاخره دقیقاً در درایه‌ای که باید، متوقف می‌شود.

برای محاسبه‌ی زمان اجرا، ابتدا به هر یک از سطرهای برنامه زمان (هزینه‌ی) اجرای آن سطر را نسبت می‌دهیم و می‌شماریم که آن سطر چند بار تکرار می‌شود. این اعداد در جدول زیر مشاهده می‌شوند. تعداد تکرار سطر ۴ وابسته به داده‌ی ورودی دارد. تعداد این تکرار را در مرحله‌ی kام t_k می‌نامیم. توجه کنید که در آن‌صورت سطر پنجم $t_k - 1$ بار تکرار می‌شود.

سطر	هزینه	تعداد
۱	c_1	n
۲	c_2	$n - 1$
۳	c_3	$n - 1$
۴	c_4	$\sum_{k=2}^{n} t_k$
۵	c_5	$\sum_{k=2}^{n}(t_k - 1)$
۶	c_6	$\sum_{k=2}^{n}(t_k - 1)$
۷	c_7	$n - 1$

حاصل‌ضرب تعداد دفعات اجرای هر سطر و زمان اجرای آن‌را برای هر سطر به‌دست آورده و جمع می‌زنیم تا زمان اجرای کل به‌دست آید:

$$T(n) = c_1 n + (c_2 + c_3 + c_7)(n - 1) + c_4 \sum_{k=2}^{n} t_k + (c_5 + c_6) \sum_{k=2}^{n} (t_k - 1)$$

مقدار ثابت‌ها به نوع سخت‌افزار، زبان برنامه‌نویسی و چنین عواملی بستگی دارد و حتا در شرایط خاص به سختی

[1] Abstraction

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

قابل محاسبه‌اند. بنابراین ما هر بار به‌جای جمع چند ثابت و عباراتی از این قبیل، ثابت جدیدی قرار می‌دهیم.

$$T(n) = An + B\sum_{k=2}^{n} t_k + C$$

۲.۱ به‌ترین و بدترین حالت و حالت متوسط

مقدار واقعی زمان اجرا به مقدار t_kها بستگی دارد. «به‌ترین حالت»[2] هنگامی اتفاق می‌افتد که آرایه از قبل مرتب شده باشد. در این حالت همواره داریم $t_k = 1$ و

$$T(n) = An + B(n-1) + C$$

که یک تابع خطی است. البته رفتار یک الگوریتم در به‌ترین حالت زیاد مهم نیست. آن‌چه مهم است رفتار الگوریتم در «بدترین حالت»[3] و «حالت متوسط»[4] است.

بیش‌ترین زمان اجرای الگوریتم هنگامی اتفاق می‌افتد که آرایه برعکس مرتب باشد. در این صورت مقدار t_k همواره برابر k است (مقایسه‌ی آخر با صفر هم حساب شده است.) برای این حالت داریم:

$$\begin{aligned} T(n) &= An + B\sum_{k=2}^{n} t_k + C \\ &= An + B\left(\frac{(n-1)(n+2)}{2} - 1\right) + C \\ &= an^2 + bn + c \end{aligned}$$

که یک تابع درجه‌ی دو برحسب n است. با توجه به این که به‌ترین و [illegible] رفتار الگوریتم متفاوتند، رفتار متوسط آن قابل توجه است.

حالت متوسط: اگر داده‌ی ورودی به‌صورت تصادفی داده شود رفتار الگوریتم (زمان اجرا) چه‌گونه است؟ (فرض می‌شود که همه‌ی حالت‌های مختلف داده‌های ورودی، احتمال برابر دارند.)

از جمله در مورد مثال فوق، $n!$ حالت (جای‌گشت) مختلف برای ترتیب n داده‌ی ورودی وجود دارد که به هر یک احتمال یکسان و برابر $\frac{1}{n!}$ منسوب می‌شود.

چیزی که در پس مفهوم حالت متوسط وجود دارد، این است که اگر چه ممکن است اجرا به اندازه‌ی زمان بدترین حالت طول بکشد، اما اگر دفعات زیادی الگوریتم بر ورودی‌های مختلف اعمال شود، به حد مذکور دست خواهیم یافت. و انتظار ما از زمان اجرا برای یک ورودی تصادفی چیست؟

[illegible] best-case[2]
worst-case[3]
average-case[4]

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

زمان اجرا در حالت متوسط در مورد مثال فوق برابر است با

$$\overline{T}(n) = An + C + \frac{1}{n!}\sum_{i=1}^{n!}\sum_{k=2}^{n} Bt_{k,i}$$

(در این حالت ما عملاً تک‌تک حالت‌ها را بررسی کرده و متوسط را یافته‌ایم.)

می‌توان ثابت کرد که کافی است به جای محاسبه‌ی مجموع فوق، از مقدار متوسط t_k یا $\bar{t}_k$ استفاده کرد. یعنی

$$\overline{T}(n) = An + C + B\sum_{k=2}^{n} \bar{t}_k$$

برای محاسبه‌ی t_k به مکانی که عنصر kام باید در آن قرار بگیرد توجه می‌کنیم که می‌تواند اولین درایه، دومین درایه، ... یا همان درایه‌ی اولیه یعنی kامین درایه باشد. هر یک از این درایه‌ها احتمال وقوع برابر دارند که به وضوح $\frac{1}{k}$ است. فرض کنید که در انتهای مرحله‌ی kام A[k] در A[i] قرار بگیرد. در این حالت $t_k = k - i + 1$ (برای $i \geq 1$). پس داریم

$$\begin{aligned}\bar{t}_k &= \sum_{i=1}^{k}\frac{1}{k}(k-i+1)\\ &= \frac{1}{k}\times\frac{k(k+1)}{2}\\ &= \frac{k+1}{2}\end{aligned}$$

پس می‌توان گفت که هر بار عضو آخر به‌طور متوسط به وسط آرایه منتقل می‌شود. و نیز داریم

$$\begin{aligned}\overline{T}(n) &= An + C + B\sum_{k=2}^{n}\frac{k+1}{2}\\ &= an^2 + bn + c\end{aligned}$$

پس رفتار حالت متوسط و بدترین حالت یکسان است.

۳.۱ مرتبه‌ی الگوریتم‌ها

معمولاً به‌ترین حالت اهمیتی ندارد. بلکه الگوریتم‌ها از نظر بدترین حالت و حالت متوسط بررسی می‌شوند. وقتی حالت متوسط و بدترین حالت مختلف باشند، دقت خاصی لازم است. برای مثال در مورد quicksort، بدترین حالت $O(n^2)$ و حالت متوسط $O(n\log n)$ است که وقتی n عدد بزرگی باشد، تفاوت عمده‌ای دارند.

آن‌چه برای ما مهم است، رفتار الگوریتم در اندازه‌ی ورودی بزرگ یا خیلی بزرگ است. در این حالت، چنان‌چه خواهیم دید، ثابت‌هایی چون c_1، c_2 و ... اهمیتی ندارند، و این با مفهوم «پیچیدگی الگوریتم»[3] نشان داده می‌شود.

برای درک به‌تر از مفهوم پیچیدگی، فرض کنید برای حل یک مسئله چهار الگوریتم داریم که بر روی n عدد کار می‌کنند. برای این الگوریتم‌ها $T(n)$ را به طور دقیق به‌دست آورده‌ایم که در جدول ۱.۱ نشان داده شده اند.

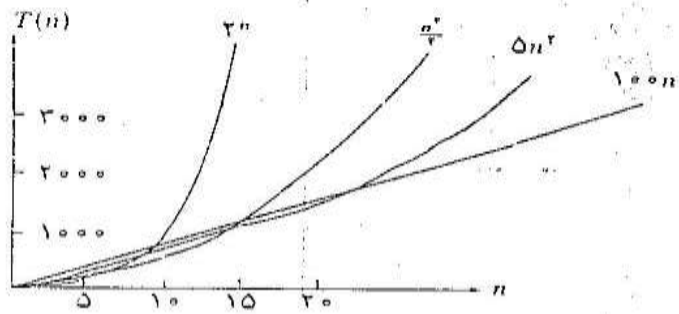
جدول فوق نشان می دهد که برای n های کوچک الگوریتم A_4 خیلی به‌تر از الگوریتم A_1 است. هم‌چنین می‌توان دید که برای الگوریتم‌های با درجه‌ی پیچیدگی زیاد، افزایش سرعت ماشین تاثیر چندانی ندارد. این موضوع‌ها را می‌توان در منحنی‌های زمان اجرای این الگوریتم‌ها برای n های مختلف در شکل ۱.۱ دید.

[3] complexity of algorithm

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

الگوریتم	$T(n)$	$O(.)$	حداکثر اندازه‌ی مسئله که در ۱۰۰۰ ثانیه حل می‌شود	حداکثر اندازه‌ی قابل حل در ۱۰۰۰ ثانیه بر روی ماشین ۱۰ برابر سریع‌تر	نسبت	نسبت اگر ماشینی ۱۰۰۰ برابرسریع‌تر انتخاب کرده باشیم
A_1	$100n$	$O(n)$	۱۰	۱۰۰	۱۰	۱۰۰۰
A_2	$5n^2$	$O(n^2)$	۱۴	۴۵	۳/۲	۱۳۱/۹۴
A_3	$n^3/2$	$O(n^3)$	۱۲	۲۷	۲/۳	۱۲۵/۹۹
A_4	2^n	$O(2^n)$	۱۰	۱۳	۱/۳	۲۰

جدول ۱.۱: مقایسه‌ی پیچیدگی الگوریتم‌ها



شکل ۱.۱: زمان‌های اجرای چهار الگوریتم برای یک مسئله.

از این مثال در می‌یابیم که الگوریتم 2^n برای nهای کوچک هم سریع‌تر است و هم آسان‌تر پیاده‌سازی می‌شود، اما برای nهای بزرگ‌تر، الگوریتم در عمل بی‌استفاده می‌شود. برای رفع این مشکل، عددهای n کوچک را در نظر نمی‌گیریم و ملاک مقایسه را nهای بزرگ‌تر می‌گیریم. نکته‌ی بسیار مهم این است که همواره عددی مثل n_0 یافت خواهد شد به‌طوری‌که برای مقادیر $n > n_0$، منحنی زمان اجرای الگوریتم 2^n از الگوریتم $100n$ جلو بزند.

۴.۱ تابع‌های رشد

برای بررسی مرتبه‌ی الگوریتم‌ها تابع‌های رشد (growth functions) استفاده می‌کنیم که با نمادهای O، Θ، Ω، o و ω نمایش داده می‌شوند. از جمله گزاره‌هایی که به چشم می‌خورد، چنین است:

- بدترین حالت الگوریتم مرتب‌ساز مبتنی بر درج، $T(n) = \Theta(n^2)$ یعنی از مرتبه‌ی دقیق n^2 است. این الگوریتم در حالت متوسط نیز $\Theta(n^2)$ است.
- الگوریتم HeapSort از $O(n \log n)$ یا مرتبه‌ی $n \log n$ است.
- کلیه‌ی الگوریتم‌های مرتب‌ساز مبتنی بر مقایسه، از مرتبه‌ی $\Omega(n \log n)$ هستند.

تعریف تابع‌های رشد

برای $g(n)$ داده‌شده، $\Theta(g(n))$ را به شکل مجموعه‌ی توابع زیر تعریف می‌کنیم:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2 > 0 \text{ and } n_0 \text{ such that } \forall n \geq n_0 \ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

عبارت $c_1 g(n) \leq f(n) \leq c_2 g(n)$ به این معنی است که درجه‌ی رشد f و g یکسان است. برای نشان دادن دقیق این مفهوم، می‌نویسیم

$$f(n) \in \Theta(g(n))$$

و یا به شکل ساده‌تر

$$f(n) = \Theta(g(n))$$

مثلاً می‌دانیم که در مورد الگوریتم مرتب‌ساز مبتنی بر درج، $T(n) = \Theta(n^2)$. زیرا می‌توان c_1 و c_2 را طوری یافت که

$$c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$$

مثال: ثابت کنید که $100n^2 + 5n - 4 = \Theta(N^2)$.
روشن‌است که اگر $c_1 = 1$ و $c_2 = 200$، داریم: $c_1 n^2 \leq 100n^2 + 5n - 4 \leq c_2 n^2$.
می‌توان ثابت کرد که هر چندجمله‌ای از مرتبه‌ی دقیق اولین جمله‌اش است، یعنی

$$a_k n^k + a_{k-1} n^{k-1} + \cdots = \Theta(n^k)$$

در مورد مرتب‌سازی مبتنی بر درج داریم:

$$\begin{aligned} T(n) &= \Theta(n^2) \\ &= \Theta(100n^2) \\ &\neq \Theta(n^3) \\ &\neq \Theta(n^2 \log n) \\ &\neq \Theta(n \log n) \end{aligned}$$

برای اثبات $T(n) \neq \Theta(n \lg n)$ می‌توان نشان داد که نمی‌توان ثابت‌های c_1 و c_2 را پیدا کرد که برای $n > n_0$ داشته‌باشیم:

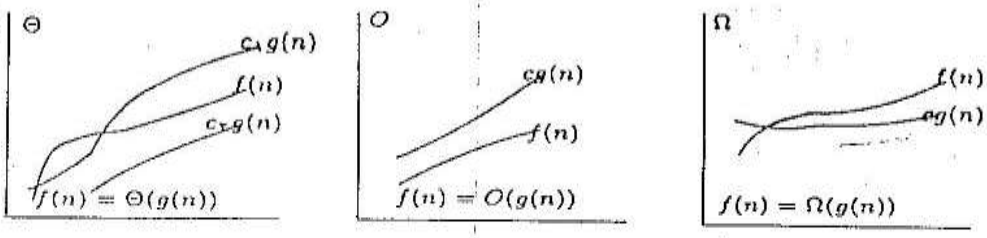
$$c_1 n \lg n \leq n^2 \leq c_2 n \lg n$$

تعریف: برای $g(n)$ داده‌شده، $O(g(n))$ را به شکل مجموعه‌ی توابع زیر تعریف می‌کنیم:

$$O(g(n)) = \{f(n) : \exists c > 0 \text{ and } n_0 \text{ such that } \forall n \geq n_0 \ 0 \leq f(n) \leq cg(n)\}$$

مثلاً در مورد مرتب‌ساز مبتنی بر درج، داریم

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲.۱: زمان‌های اجرای چهار الگوریتم برای یک مسئله.

$$
\begin{aligned}
T(n) &= O(n^{۲}) \\
&= O(۱۰۰n^{۲}) \\
&\neq O(n^{۳}) \\
&\neq O(n^{۲} \lg n) \\
&\neq O(n \lg n) \\
&\neq O(n)
\end{aligned}
$$

این نکته جالب است که می‌توان گفت مسئله از $O(n^{۱۰۰})$ است، ولی این اطلاع چندان مفید نیست. به‌همین جهت باید در حالت‌هایی که محاسبه‌ی Θ مشکل است، باید تابع داخل پرانتز O را تا حد امکان کوچک‌تر کنیم.

تعریف: برای $g(n)$ داده‌شده، $\Omega(g(n))$ را به شکل مجموعه‌ی توابع زیر تعریف می‌کنیم:

$$\Omega(g(n)) = \{f(n) : \exists c > ۰ \text{ and } n_۰ \text{ such that } \forall n \geq n_۰ \; ۰ \leq cg(n) \leq f(n)\}$$

مثلاً برای مرتب‌سازی مبتنی بر درج داریم

$$
\begin{aligned}
T(n) &= \Omega(n \log n) \\
&= \Omega(n) \\
&= \Omega(n^{۲}) \\
&\neq \Omega(n^{۲} \log n)
\end{aligned}
$$

در عمل نماد Ω برای نشان دادن حد پایین یک تابع دیگر به‌کار می‌رود.

تابع Θ در واقع عطف O و Ω است یعنی

$$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مفهوم تابع‌های رشد تعریف شده در شکل ۲.۱ نشان داده‌شده است.
دو نماد زیر نیز در همین رابطه تعریف شده‌اند:

تعریف:

$$o(g(n)) = \{f(n) \mid \text{for any positive constant } c > 0, \exists n_0 > 0, \text{ such that } \forall n > n_0 : 0 \le f(n) < cg(n)\}$$

این رابطه نزدیکی زیادی با O دارد بااین تفاوت که دیگر $f(n)$ نمی‌تواند با $cg(n)$ برابر باشد و باید الزاماً کوچکتر باشد.

تعریف:

$$\omega(g(n)) = \{f(n) \mid \text{for any positive constant } c > 0, \exists n_0 > 0, \text{such that } 0 \le cg(n) < f(n)\}$$

رابطه‌ی ω با Ω مثل o با O است.
نتیجه‌ای که از تعریف‌های فوق به‌دست می‌آید را در عبارت‌های زیر (هرچند نادقیق از نظر ریاضی) خلاصه می‌کنیم:
اگر f درجه‌ی رشد F و g درجه‌ی رشد G باشد.

$$\left|\begin{array}{lcl} F = \Theta(g) & \Longleftrightarrow & f = g \\ F = O(g) & \Longleftrightarrow & f \le g \\ F = o(g) & \Longleftrightarrow & f < g \\ F = \Omega(g) & \Longleftrightarrow & f \ge g \\ F = \omega(g) & \Longleftrightarrow & f > g \end{array}\right|$$

مثالی دیگر

الگوریتم مرتب‌سازی مبتنی بر ادغام (MergeSort) را در نظر می‌گیریم.

```
Merge_Sort(A, p, r):
   Input:  A (the array to sort)
           p (starting index)
           r (ending index)
1  if p < r
2     then q := (p + r) div 2
3          Merge_Sort(A, p, q)
4          Merge_Sort(A, q + 1, r)
5          Merge(A, p, q, r)
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

این الگوریتم، هر بار آن بخش از آرایه را که می‌خواهد مرتب کند، ابتدا نصف می‌کند. هر نیمه را به‌صورت بازگشتی مرتب می‌کند و با ادغام دو بخش مرتب‌شده، کار را تمام می‌کند.

برای ادغام دو لیست مرتب، ساده‌ترین الگوریتم به این شکل است که دو اشاره‌گر را باید بر ابتدای لیست‌ها قرار داد و هربار عدد کمتر را به لیست خروجی افزود. اگر طول دو لیست اول m و n باشد، زمان مسئله $O(m+n)$ است. منتها این مسئله از $O(m+n)$ فضای اضافی می‌برد.

ادغام کردن «درجا»، یا in-place merging نوعی ادغام است که از فضای اضافی برای ادغام استفاده نمی‌کند. در واقع فضای اضافی مورد نیاز از $O(1)$ می‌شود.

اثبات درستی الگوریتم، به شکل استقرایی صورت می‌گیرد، یعنی برای پایه با $n = 1$ اثبات مسئله بدیهی است و برای $n > 1$ درستی آن به‌سادگی از فرض استقرا اثبات می‌شود (فرض می‌کنیم برای $n/2$ اثبات صورت گرفته است).

برای این الگوریتم داریم:

$$T(n) = \begin{cases} a & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + bn & n > 2 \end{cases}$$

که می‌توان (بعداً می‌بینیم) ثابت کرد که

$$T(n) = \Theta(n \log n).$$

۵.۱ روش‌های تحلیل الگوریتم‌ها

الگوریتم‌ها به‌دو دسته‌ی اصلی تقسیم می‌شوند: الگوریتم‌های ترتیبی و بازگشتی.

۱.۵.۱ الگوریتم‌های ترتیبی

برای به‌دست آوردن زمان اجرای یک الگوریتم ترتیبی فرض می‌کنیم که اجرای یک دستور ساده (Statement) زمان ثابتی نیاز دارد. بنابراین اگر تعداد ثابتی دستور ساده پشت سر هم باشند، در مجموع زمانی ثابت می‌گیرند. $T(n)$ زمان اجرای یک تکه‌برنامه به صورت‌های زیر محاسبه می‌شود:

- اگر تکه‌برنامه‌ی مورد نظر خود شامل k تکه‌برنامه با زمان‌های اجرای $T_1(n) = O(f_1(n))$، $T_2(n) = O(f_2(n))$ تا $T_k(n) = O(f_k(n))$ باشد، در آن‌صورت

$$T(n) = \sum_{i=1}^{k} T_i(n) = O(\max_{1 \le i \le k} (f_i(n)))$$

زمان اجرای یک برنامه شامل چند تکه برنامه = O(زمان اجرای بزرگترین تکه برنامه)

- اگر تکه‌برنامه ساختار حلقه داشته باشد (شامل for، while، repeat و امثال آن[1]) و $g(n)$ بار تکه‌برنامه‌ای دیگر با زمان اجرای $S(n) = O(f(n))$ را تکرار کند،

[1] بعضی از مواقع حلقه با goto هم ساخته می‌شود که تشخیص این موارد نیاز به دقت دارد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$T(n) = g(n)S(n) = O(g(n)f(n))$$

- اگر ساختار if باشد و قسمت‌های then و else آن به ترتیب زمان‌های $T_1(n) = O(f_1(n))$ و $T_2(n) = O(f_2(n))$ را داشته باشند،

$$T(n) = \max\{T_1(n), T_2(n)\} = O(\max\{f_1(n), f_2(n)\})$$

به‌طور شهودی، مرتبه‌ی زمان اجرای یک الگوریتم، همان مرتبه‌ی تکه‌ای است که در برنامه بیش‌ترین زمان را می‌گیرد. چرا که ما همیشه برای تابع رشد بدترین حالت را در نظر می‌گیریم.

به این مثال برای الگوریتم Bubble Sort توجه کنید:

```
For i:=1 to n-1 do
For j:=n downto i+1 do
  If A[j] > A[j-1]
  Then Swap ( a[j] , A[j-1] )
```

توجه کنید که زمان مقایسه و تعویض را ثابت در نظر می‌گیریم. تعداد دقیق مقایسه‌ها و حداکثر تعداد تعویض‌ها برابر است با

$$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} O(1) = \Theta(n^2)$$

این الگوریتم در بدترین حالت بررسی شده است[7]. البته با توجه به مطالب درس همین موضوع را بدون محاسبه می‌توان فهمید. چون زمان مقایسه و تعویض را ثابت در نظر گرفتیم، زمان if یک عدد ثابت است. تکرار حلقه داخلی در بدترین حالت از مرتبه‌ی n است و بنابراین کل آن از نیز از مرتبه n می شود. تکرار حلقه بیرونی نیز در بدترین حالت از مرتبه‌ی n است و چون کل حلقه‌ی داخلی از مرتبه‌ی n بود، بنا به قوانین گفته شده زمان کل حلقه از مرتبه‌ی حاصل ضرب آن دو یعنی n^2 است.

۶.۱ الگوریتم‌های بازگشتی

معمولاً الگوریتم‌های بازگشتی مسئله را به دو (یا چند) زیرمسئله‌ی کوچکتر تقسیم می‌کند و آن‌ها را به‌صورت بازگشتی و با همان الگوریتم حل و نتایج آن‌ها با با هم ترکیب می‌کند. بنابراین زمان اجرای راه حل، حاصل جمع زمان حل زیر مسئله‌ها به علاوه‌ی زمان لازم برای شکستن مسئله به زیر مسئله و نیز ترکیب جواب‌های آن‌هاست.

[7] به‌دست آوردن تعداد تعویض ها در حالت متوسط کار چندان ساده‌ای نیست و نیاز به بحث آماری دارد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱.۶.۱ مسئله‌ی برج‌های هانوی

حداقل تعداد حرکت در این مسئله وقتی است که بزرگ‌ترین سکه مستقیماً به میله‌ی مقصد منتقل شود. برای این کار لازم است که بقیه‌ی سکه‌ها روی میله‌ی دوم باشند که این کار به‌صورت بازگشتی انجام می‌شود. پس از انتقال سکه‌ی بزرگ‌تر، بقیه‌ی سکه‌ها (۱ − n عدد) به صورت بازگشتی به میله‌ی مقصد منتقل می‌شوند. توجه کنید که با این الگوریتم کلیه‌ی قواعد بازی رعایت شده و نمی‌توان الگوریتمی با تعداد حرکت کم‌تر برای این مسئله پیدا کرد. با این توضیح، اگر $T(n)$ کمینه‌ی تعداد حرکت‌ها برای n سکه باشد، داریم

$$T(n) = \begin{cases} ۱, & n = ۱ \\ T(n-۱) + ۱ + T(n-۱), & n > ۱ \end{cases}$$

که این یک «رابطه‌ی بازگشتی» (Recurrence Relation) است.

برای مثال دیگر به زمان الگوریتم مرتب‌ساز مبتنی بر ادغام توجه کنید:

$$T(n) = \begin{cases} ۰, & n = ۱ \\ T(\lfloor n/۲ \rfloor) + T(\lceil n/۲ \rceil) + Cn, & n > ۱ \end{cases}$$

اگر $n = ۲^k$، داریم،

$$T(n) = \begin{cases} ۰, & n = ۱ \\ T(n) = ۲T(n/۲) + Cn, & n > ۱ \end{cases}$$

در این مثال Cn هزینه ادغام است و $T(\lfloor n/۲ \rfloor)$ و $T(\lceil n/۲ \rceil)$ هزینه‌های حل زیر مسئله‌ها بنا به استقرا هستند.

۷.۱ روش‌های حل رابطه‌های بازگشتی

رابطه‌های بازگشتی را می‌توان به روش‌های زیر حل کرد.

۱. استقرا: حدس خوب و استقرا

۲. تکرار با جای‌گذاری: بازکردن فرمول و به‌دست آوردن جواب به طور صریح

۳. قضیه‌ی اصلی

۴. روش‌های حل رابطه‌های بازگشتی همگن و ناهمگن

۱.۷.۱ حدس و استقرا

در این روش ابتدا سعی می‌کنیم جواب را حدس بزنیم و در مرحله‌ی بعد آن را اثبات می‌کنیم. برای حدس جواب ممکن است از قضیه‌ی اصلی استفاده کنیم.

اگر بخواهیم اثبات کنیم که $T(n) = O(f(n))$، باید با استقرا اثبات کنیم که $T(n) \le cf(n)$ و در مراحل استقرا نشان دهیم که حداقل یک مقدار ثابت برای c وجود دارد که در این نامعادله صدق کند. برای اثبات $T(n) = \Omega(f(n))$، باید به‌همین ترتیب رابطه‌ی $T(n) \ge cf(n)$ و برای اثبات $T(n) = \Theta(f(n))$، باید هم $T(n) = O(f(n))$ و هم $T(n) = \Omega(f(n))$ را اثبات کنیم.

مثال ۱

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$
$$T(2) = T(1) = O(1)$$

نکته: در حل رابطه‌های بازگشتی برای به‌دست آوردن مرتبه با $O()$ از $\lfloor .. \rfloor$ و $\lceil .. \rceil$ می توان صرف‌نظر کرد. البته در صورتی‌که از این علایم صرف‌نظر نکنیم جواب دقیق‌تری بدست خواهیم آورد.

حل مثال با توجه به این که هربار مسئله را به دو زیر مسئله با اندازه‌ی نصف تقسیم می‌کنیم، بعد از $\lg n$ بار مسئله‌ای با اندازه‌ی واحد خواهیم داشت و چون زیر مسئله‌ها را با $O(n)$ باهم ترکیب می‌کنیم، حدس می‌زنیم که کل مسئله از مرتبه‌ی $O(n \lg n)$ است. باید ثابت کنیم که عدد مثبت c وجود دارد به‌طوری‌که $T(n) \le cn \lg n$

پایه‌ی استقرا:

$$n = 2 \Rightarrow T(2) \le 2c \Rightarrow c \ge \frac{T(2)}{2} > 0$$

فرض استقرا: برای $k < n$ فرض می‌کنیم $T(k) \le ck \lg k$

حکم استقرا: باید ثابت کنیم: $T(n) \le cn \lg n$. داریم:

$$\begin{aligned} T(n) &\le 2c\lfloor \frac{n}{2} \rfloor \lg \lfloor \frac{n}{2} \rfloor + n \\ &\le cn \lg \frac{n}{2} + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n \end{aligned}$$

اگر $n \ge 1$، حکم اثبات می‌شود. نکته‌ی مهم در این روش حل آن است که باید در حکم استقرا دقیقاً به همان فرمولی برسیم که حدس زده‌ایم (مثلاً $\le cn \lg n$). در صورتی‌که این امر اثبات نشود، باید حدس اولیه را تصحیح کرد.

مثال ۲

برای مثال دیگر به تحلیل مرتب‌سازی متنی برادغام توجه کنید. فرض می‌کنیم $n = 2^k$.

$$T(2) = a$$
$$T(n) = 2T(\frac{n}{2}) + bn$$

حدس: $T(n) \le cn \lg n \Rightarrow T(n) = O(n \lg n)$

پایه: $T(2) = a \le 2c$

فرض استقرا: برای $\frac{n}{2}$ داریم، $T(n/2) \le c\frac{n}{2} \lg \frac{n}{2}$ آن‌گاه

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2}\lg\frac{n}{2} + bn \\ &= cn\lg n + (b-c)n \leq cn\lg n \\ &\leq cn\log n, if\, b - c \leq 0 \text{ and } c \geq b, c \geq a/2 \end{aligned}$$

این اثبات فقط بیان می دارد که به‌ازای $n = 2^k$ الگوریتم از $O(n\lg n)$ است امّا توجه به این نکته ضروری است که برای هر عدد دلخواه n دو توان متوالی دو را می‌توان یافت که عدد مذکور بین آن دو باشد($2^k \leq n < 2^{k+1}$). چون تابع رشد تابعی صعودی است می توان اثبات کرد که $T(n)$، مرتبه‌ی الگوریتم برای n، همان مرتبه‌ی $T(2^k)$ است.

تمرین: برای تابع

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor + 17) + n, \quad n > 100 \\ T(1) &= T(2) = \ldots = T(100) = 1 \end{aligned}$$

حدس می‌زنیم: $T(n) \leq cn\lg n$ (در واقع $T(n) = O(n\lg n)$) این حدس را ثابت کنید.

مثال ۳

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1$$

حدس می‌زنیم تابع رشد از مرتبه $O(n)$ است. باید ثابت کنیم $T(n) \leq cn$. اگر از استقرا برای اثبات استفاده کنیم، در نهایت خواهیم داشت،

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + c\lceil \frac{n}{2} \rceil + 1 = cn + 1 \geq Cn$$

بنابراین حدس اشتباه است که به این صورت آن را تصحیح می‌کنیم: $T(n) \leq cn + b$. حال داریم:

$$T(n) \leq c\lfloor \frac{n}{2} \rfloor + c\lceil \frac{n}{2} \rceil + 2b + 1 = cn + 2b + 1 \leq Cn + B \Rightarrow B \leq -1$$

که اثبات کامل می‌شود.

مثال ۴

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$$

حدس می‌زنیم تابع رشد از مرتبه‌ی n باشد. باید $T(n) \leq cn$. با استفاده از استقرا داریم:

$$T(n) \leq 2C\lfloor \frac{n}{2} \rfloor + n \leq cn + n = (c+1)n \geq Cn$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

در مورد این مثال حدس اولیه غلط است. ولی از روی نتیجه‌ی به‌دست آمده متوجه می‌شویم که مرتبه‌ی الگوریتم بالاتر از n است. حدس جدید $T(n) \leq cn \log n + b$ را اگر امتحان کنیم درست در می‌آید.

یکی از راه‌های حل بعضی از رابطه‌های بازگشتی تغییر متغیر است. به مثال زیر توجه کنید.

مثال ۵

$$T(n) = 2T(\sqrt{n}) + \log_2 n$$

متغیر جدید m را به‌صورت $m = \log_2 n$ در نظر می‌گیریم. در نتیجه خواهیم داشت، $T(2^m) = 2T(2^{\frac{m}{2}}) + m$. و یا

$$S(m) = 2S(\frac{m}{2}) + m$$

با توجه به مثال‌های قبلی داریم.

$$S(m) = O(m \log m) \Rightarrow T(n) = O(\log n \log \log n)$$

مثال ۶

می خواهیم با کم‌ترین تعداد مقایسه‌های بین عناصر یک آرایه‌ی nتایی A، کوچک‌ترین و بزرگ‌ترین عدد آرایه را به‌دست آوریم. می‌توان اثبات کرد که کمترین تعداد مقایسه‌های ممکن $\lceil \frac{3n}{2} \rceil - 2$ است[8].

روش غیر بازگشتی اول: وقتی بزرگ‌ترین عنصر به‌دست آمد نیاز به مقایسه‌ی آن با کوچک‌ترین عنصری که تا کنون به‌دست آمده نیست. پیدا کردن بزرگترین n عنصر $n - 1$ مقایسه لازم دارد. برای محاسبه‌ی کوچک‌ترین عنصر $n - 2$ مقایسه دیگر کافی است. پس این کار با $2n - 2$ مقایسه ممکن است.

روش غیر بازگشتی دوم: برای $n = 2k$ با یک مقایسه بین هر دو عنصر متوالی Minها و Maxهای آن‌ها را پیدا می‌کنیم. با مقایسه‌ی همه‌ی Minها کوچک‌ترین عنصر و با مقایسه‌ی همه‌ی Maxها بزرگ‌ترین عنصر را به‌دست می‌آوریم. این روش برای $n = 2k$ بهینه است و به $n/2 + (n/2 - 1) + (n/2 - 1) = 3n/2 - 2$ مقایسه نیاز دارد. تعداد مقایسه‌های این روش برای $n = 2k + 1$ برابر $3k = \lceil 3n/2 \rceil$ است و بهینه نیست.

در روش بازگشتی آرایه را به دو قسمت تقریباً مساوی تقسیم می‌کنیم. در هر قسمت Min و Max را به‌دست می‌آوریم. سپس دو Min را با هم و دو Max را با هم مقایسه می‌کنیم تا بزرگ‌ترین و کوچک‌ترین عنصر به‌دست آید.

[8] اثبات این مطلب خیلی ساده نیست!

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Procedure Maximum (A: array of integer; head, tail : integer;
                   var max, min : integer);
var
  max2, min2 : integer;
begin
  if head = tail then begin
     max := A[head];
     min := max;
  end else if tail - head = 1 then begin
     if A[head] > A[tail] then begin
        max := A[head];
        min := A[tail];
     end else begin
        max := A[tail];
        min := A[head];
     end;
  end else begin
     Maximum(A, head, (head+tail) div 2, max, min);
     Maximum(A, (head+tail) div 2+1, tail, max2, min2);
     if max2 > max then max := max2;
     if min2 < min then min := min2;
  end;
```

اگر تعداد مقایسه‌های عنصرهای آرایه در این روش $T(n)$ باشد، داریم،

$$T(n) = \begin{cases} ۰ & n = ۱ \\ ۱ & n = ۲ \\ T(\lfloor n/۲ \rfloor + T(\lceil n/۲ \rceil) + ۲ & n > ۲ \end{cases}$$

این راه‌حل نیز بهینه نیست، چون مثلاً $T(۶) = ۸$ در حالی که می‌توان با ۷ مقایسه $(۲ - \lceil \frac{۱۸}{۳} \rceil)$ این کار را انجام داد. البته راه‌حل فوق برای $n = ۲^k$ بهینه است و با استفاده از استقرا به‌سادگی می‌توان آن‌را نشان داد.

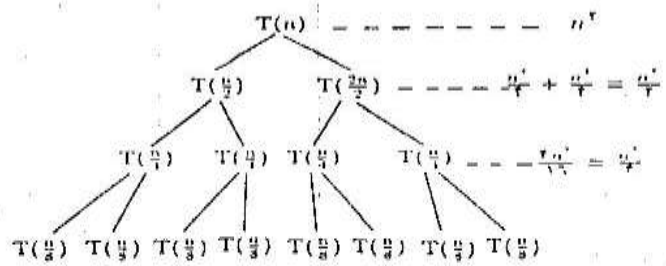
$$T(۲^k) = ۲[۳ \times ۲^{k-۲} - ۲] - ۲ = ۳ \times ۲^{k-۱} - ۲$$

راه‌حل زیر برای این مسئله بهینه است. آرایه را به دو قسمت با اندازه‌ی ۲ و $n - ۲$ تقسیم کن. با $T(n - ۲) + ۱$ مقایسه کوچکترین و بزرگترین عنصرهای هر قسمت را پیدا کن و با ۲ مقایسه‌ی دیگر جواب را به دست آور. در این صورت،

$$T(n) = \begin{cases} ۰ & n = ۱ \\ ۱ & n = ۲ \\ T(n - ۲) + ۳ & n > ۲ \end{cases}$$

که جواب آن همان مقدار بهینه است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳.۱: درخت بازگشت برای مثال ۱

۲.۷.۱ تکرار با جای‌گذاری

در این روش رابطه‌ی بازگشتی را آن قدر بسط می‌دهیم تا به جواب نهایی برسیم.

مثال ۱: $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$

$$\begin{aligned}
T(n) &= 3T(\lfloor \tfrac{n}{4} \rfloor) + n \\
&= 3^2 T(\lfloor \tfrac{\lfloor n/4 \rfloor}{4} \rfloor) + 3\lfloor \tfrac{n}{4} \rfloor + n = 3^2 T(\lfloor \tfrac{n}{16} \rfloor) + 3\lfloor \tfrac{n}{4} \rfloor + n \\
&= 3^3 T(\lfloor \tfrac{n}{64} \rfloor) + 3^2 \lfloor \tfrac{n}{16} \rfloor + 3\lfloor \tfrac{n}{4} \rfloor + n \\
&= \dots \\
&= \dots \\
&\le 3^i T(\tfrac{n}{4^i}) + n \textstyle\sum_{j=0}^{i-1} (\tfrac{3}{4})^j
\end{aligned}$$

از برابری $\lfloor \frac{\lfloor n/4 \rfloor}{4} \rfloor = \lfloor \frac{n}{16} \rfloor$ در فرمول‌های فوق استفاده می‌کنیم.

برای حل رابطه باید آن را باز کنیم تا به $T(1)$ برسیم. داریم $\frac{n}{4^i} = 1 \Rightarrow i = \log_4 n$، بنابراین

$$T(n) \le 3^{\log_4 n} \cdot T(1) + n \sum_{j=0}^{\log_4 n - 1} (\tfrac{3}{4})^j$$

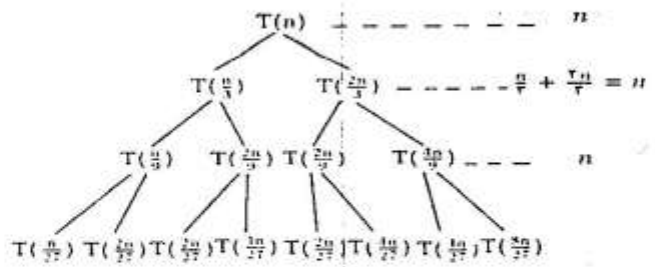
برای n های بزرگ این مجموع مقداری ثابت است. لذا، $T(n) \le C_1 n^{\log_4 3} + C_2 n \Rightarrow$. اما می‌دانیم که

$$\lim_{n \to \infty} \sum_{j=0}^{\log_4 n - 1} (\tfrac{3}{4})^j = 4 \Rightarrow C_2 < 4$$

و داریم، $3^{\log_4 n} = n^{\log_4 3}$. لذا

$$T(n) \le Cn^{\log_4 3} + 4n \Rightarrow T(n) = O(n)$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴.۱: درخت بازگشت برای مثال ۲

در اکثر موردها روش بسط فرمول جواب می‌دهد ولی در بعضی حالات نمی‌توان از باز کردن فرمول به جایی رسید. مثلاً، رابطه‌ی بازگشتی برای تولید عددهای فیبوناچی از این نوع است: $F(n) = F(n-1) + F(n-2)$ که داریم $F(1) = F(2) = 1$. برای حل رابطه‌های مانند این، از روش حل روابط بازگشتی همگن استفاده می‌شود.

درخت بازگشت

یکی از روش های خوب برای حل یا حدس رابطه‌های بازگشتی از طریق تکرار استفاده از درخت بازگشت است.

مثال ۱: $T(n) = 2T(\frac{n}{2}) + n^2$

$$\begin{aligned} T(n) &= n^2 + \frac{n^2}{4} + \ldots + \frac{n^2}{2^i} + \ldots + \frac{n^2}{2^{\log_2 n}} \\ &= n^2(1 + \frac{1}{2} + \ldots + \frac{1}{2^i} + \ldots + \frac{1}{2^{\log_2 n}}) \\ &\le 2n^2 \\ \Rightarrow T(n) &= O(n^2) \end{aligned}$$

توجه: آخرین سطح این درخت دارای n برگ با مقدارهای ۱ خواهد بود.

مثال ۲ $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n$

این درخت به‌گونه‌ای رشد می‌کند که در نهایت بعد از k مرحله به $T(1)$ می‌رسیم. در اینجا k همان ارتفاع بیشینه‌ی درخت است. این درخت متوازن نیست و در سمت چپ ارتفاع کمتری دارد. برای به‌دست آوردن ارتفاع بیشینه‌ی درخت ، داریم، $(\frac{2}{3})^i n = 1$، پس $i = \log_{\frac{3}{2}} n$، لذا $T(n) \le n \log_{\frac{3}{2}} n$ از طرف دیگر داریم، $\log_{\frac{3}{2}} n = O(\log_2 n)$. در نتیجه $T(n) = O(n \log_2 n)$.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۳.۷.۱ قضیه‌ی اصلی

در این‌جا ما خود قضیه‌ی اصلی را اثبات نمی‌کنیم ولی می‌توان آن را از روش جای‌گذاری اثبات کرد.

برای $a \geq 1, b > 1$ و تابع $f(n)$ حل رابطه‌ی بازگشتی $T(n) = aT(\frac{n}{b}) + f(n)$ (که در آن $\frac{n}{b}$ می‌تواند به‌صورت کف یا سقف باشد) به‌قرار زیر است:

الف — اگر $f(n) = O(n^{\log_b a - \epsilon})$، برای $\epsilon > 0$ (یعنی رشد تابع $f(n)$ از تابع $n^{\log_b a}$ به‌صورت چند جمله‌ای کم‌تر باشد.) در این صورت، $T(n) = \Theta(n^{\log_b a})$

ب — اگر $f(n) = \Theta(n^{\log_b a})$، دراین صورت، $T(n) = \Theta(n^{\log_b a} \log_2 n)$

ج — اگر $f(n) = \Omega(n^{\log_b a + \epsilon})$. دراین صورت، $T(n) = \Theta(f(n))$.

به‌عبارت ساده‌تر ولی نادقیق، قضیه‌ی اصلی را می‌توان به‌صورت زیر بیان کرد. اگر G درجه‌ی رشد تابع $g(n) = n^{\log_b a}$ و F درجه‌ی رشد تابع $f(n)$ باشد، خواهیم داشت:

۱. اگر $F > G$، $T(n) = \Theta(f(n))$

۲. اگر $G > F$، $T(n) = \Theta(g(n))$

۳. اگر $F = G$، $T(n) = \Theta(g(n) \log_2 n)$.

توجه: $>$ و $<$ به‌معنی بزرگ‌تر یا کوچک‌تر چند جمله‌ای است. در صورتی‌که هیچ‌کدام از حالت‌های بالا وجود نداشته باشد نمی‌توان قضیه‌ی اصلی را اعمال کرد و باید از روش‌های دیگر استفاده کرد.

مثال ۱: $T(n) = 9T(\frac{n}{3}) + n$

داریم، $f(n) = n$ و $g(n) = n^{\log_3 9} = n^2$. بدیهی است که درجه‌ی رشد n^2 از n به‌صورت چند جمله‌ای بیشتر است. لذا،

$$f(n) = O(n^{2-1}) \Rightarrow T(n) = \Theta(n^2)$$

مثال ۲: $T(n) = T(\frac{2n}{3}) + 1$

داریم، $g(n) = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$ و $f(n) = 1$. پس، $T(n) = \Theta(\log_2 n)$.

مثال ۳: $T(n) = 3T(\frac{n}{4}) + n \log_2 n$

چون، $f(n) = n \log_2 n$ و $g(n) = n^{\log_4 3} = n^{0/793}$ و $f(n) = \Omega(n^{0/793 + 0/1})$. داریم $T(n) = \Theta(n \log_2 n)$.

مثال ۴: $T(n) = 2T(\frac{n}{2}) + n \log_2 n$

برای این مثال، $g(n) = n^{\lg 2} = n$ و $f(n) = n \lg_2 n = (n)$ ولی اختلاف به‌صورت نمایی نیست یعنی هیچ ϵ ای نمی‌توان پیدا کرد که برای کلیه‌ی nهای بزرگ، رابطه‌ی $n \log_2 n < n^{1+\epsilon}$ برقرار باشد. بنابراین این مسئله را باید از روش دیگری، مثلاً استقرا حل کرد. جواب $T(n) = O(n \log^2 n)$ است که با استقرا اثبات می‌شود.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۴.۷.۱ رابطه‌های بازگشتی همگن[1]

هدف این مقاله معرفی و بررسی رابطه‌های بازگشتی همگن و ارایه‌ی روش‌های حل آن‌هاست. به مثال‌های زیر توجه کنید:

مثال ۱. به چند طریق می‌توان صفحه‌ای $n \times ۲$ را با موزاییک‌های ۱×۲ فرش کرد؟

حل: اگر جدول $n \times ۲$ را بتوان با f_n روش مختلف با موزاییک‌های $n \times ۲$ فرش کرد به‌راحتی می‌توان ثابت کرد که $f_n = f_{n-۱} + f_{n-۲}$. به‌این ترتیب که اگر در ستون nام جدول یک موزاییک ۱×۲ به‌صورت عمودی گذاشته شود، جدولی $(n-۱) \times ۲$ می‌ماند که باید با موزاییک‌های ۱×۲ فرش شود که به $f_{n-۱}$ طریق ممکن است. در غیر این صورت دو ستون n و $n-۱$ام با دو موزاییک ۱×۲ افقی پوشانده شده‌اند، که جدولی $(n-۲) \times ۲$ می‌ماند که باید با موزاییک‌های ۱×۲ فرش شود که به $f_{n-۲}$ طریق ممکن است. پس در کل تعداد روش‌های فرش کردن جدول $n \times ۲$ با موزاییک‌های ۱×۲ $f_{n-۱} + f_{n-۲}$ است، یعنی $f_n = f_{n-۱} + f_{n-۲}$. این رابطه، دنباله‌ی معروفی را مشخص می‌کند که با مقادیر اولیه‌ی $f_۰ = ۰$ و $f_۱ = ۱$ دنباله‌ی فیبوناچی نام دارد.

تمرین ۱ به چند طریق می‌توان صفحه‌ای $۳ \times n$ را با موزاییک‌های ۱×۲ فرش کرد؟ (رابطه‌ای بازگشتی را بیابید و آن را حل کنید.)

دقت می‌کنیم که رابطه‌ی $f_n = f_{n-۱} + f_{n-۲}$ هر عضو دنباله را بر حسب اعضای قبلی دنباله بیان می‌کند، مثلاً برای پیدا کردن $f_۲$ می‌توان از رابطه‌ی $f_۲ = f_۰ + f_۱$ استفاده کرد و به‌دست آورد: $f_۲ = ۰ + ۱ = ۱$. حال به‌همین ترتیب با جمع $f_۱$ و $f_۲$ می‌توان $f_۳$ را به دست آورد. ضمناً برای ساختن دنباله باید دو عضو اول آن را داشته باشیم تا با استفاده از آنها $f_۲$ و $f_۳$ و همین‌طور تا f_n مشخص شوند. حل رابطه‌ی بازگشتی به‌این معناست که ما هر عضو دنباله‌ی f_n را بدون استفاده از اعضای قبلی و مستقل از آن اعضا بیان کنیم.

اگر c_iها عددهای حقیقی باشند، به رابطه‌ی بازگشتی زیر رابطه‌ی بازگشتی همگن از درجه‌ی k می‌گویند:

$$a_n = c_۱ a_{n-۱} + c_۲ a_{n-۲} + \cdots + c_k a_{n-k} \qquad (۱)$$

در این رابطه a_n بر حسب k جمله‌ی قبل آن داده شده است و برای ساخت دنباله به k جمله‌ی اول آن احتیاج داریم. تابع $g(n)$ را یک جواب دنباله‌ی بازگشتی فوق می‌نامند. اگر دنباله‌ی $a_n = g(n)$ $(n \in \mathbb{N})$ در رابطه‌ی بازگشتی صدق کند.

قضیه ۱. اصل برهم نهی: اگر $g_i(n)$ $(i = ۱, \ldots, r)$ جوابی برای

$$a_n = c_۱ a_{n-۱} + c_۲ a_{n-۲} + \cdots + c_k a_{n-k} + f_i(n) \qquad (۲)$$

باشند، آنگاه هر ترکیب خطی از این k جواب به صورت $A_۱ g_۱(n) + A_۲ g_۲(n) + \cdots + A_r g_r(n)$ که در آن A_iها اعدادی حقیقی‌اند، پاسخی برای رابطه‌ی بازگشتی شماره‌ی ؟؟ است. به‌ویژه، چون در رابطه‌های بازگشتی همگن $f_i(n) = ۰$، هر ترکیب خطی جواب‌های یک رابطه‌ی بازگشتی همگن باز یک جواب همان رابطه‌ی بازگشتی است.

اثبات: فرض کنیم، $h(n) = A_۱ g_۱(n) + A_۲ g_۲(n) + \cdots + A_r g_r(n)$؛ چون $g_i(n)$ یک جواب $a_n = c_۱ a_{n-۱} + c_۲ a_{n-۲} + \cdots + c_k a_{n-k} + f_i(n)$ است، پس داریم:

$$g_i(n) = c_۱ g_i(n-۱) + c_۲ g_i(n-۲) + \cdots + c_k g_i(n-k) + f_i(n).$$

[1] این بخش توسط آقای وهاب مهرکنی دانش‌جوی سابق دانشکده تهیه شده است. با تشکر از ایشان.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

پس نتیجه می‌شود:

$$h(n) = c_1 h(n-1) + c_2 h(n-2) + \cdots + c_k h(n-k) + A_1 f_1(n) + \cdots + A_r f_r(n)$$

بنابراین حکم قضیه ثابت است.

حل رابطه‌های بازگشتی همگن

رابطه‌های بازگشتی همگن حل ساده‌ای دارند که بدین صورت است: اگر $g(n) = x^n$، جواب رابطه‌ی بازگشتی همگن شماره‌ی ۱ باشد، داریم:

$$x^n - c_1 x^{n-1} - c_2 x^{n-2} - \cdots - c_k x^{n-k} = 0$$

یا به عبارت دیگر،

$$x^k - c_1 x^{k-1} - \cdots - c_k = 0 \qquad (3)$$

است، یعنی x جواب معادله درجه k فوق است. این معادله را **معادله‌ی سرشت‌نما یا معادله‌ی متشکله**[10] رابطه‌ی بازگشتی می‌نامیم. اگر x_i، ریشه‌ی معادله‌ی سرشت‌نما باشد، بدیهی است که $a_n = x_i^n$ یک جواب رابطه‌ی بازگشتی است و بنا بر قضیه‌ی قبلی هر ترکیب خطی از x_i^n ها هم یک جواب رابطه‌ی بازگشتی است. ضمناً این جواب‌ها پایه‌ای برای مجموعه جواب این رابطه می باشند. البته چگونگی تحقیق این مطلب به مقدماتی از جبر خطی احتیاج دارد. با اثبات این مطلب می‌توان گفت: تمام جواب‌های رابطه‌ی ۱ به‌صورت ترکیبی خطی از x_i^n ها است. (چگونگی اثبات را تحقیق کنید.) به طور مثال ترکیب خطی:

$$a_n = t_1 x_1^n + t_2 x_2^n + \cdots + t_k x_k^n; \qquad (4)$$

را در نظر می‌گیریم. اگر در این رابطه‌ی بازگشتی k عنصر اول این دنباله داده شده باشند، باید k معادله‌ی زیر برقرار باشد:

$$\begin{cases} a_0 = t_1 + t_2 + \cdots + t_k \\ a_1 = t_1 x_1 + t_2 x_2 + \cdots + t_k x_k \\ \vdots \\ a_{k-1} = t_1 x_1^{k-1} + t_2 x_2^{k-1} + \cdots + t_k x_k^{k-1} \end{cases}$$

این یک دستگاه k معادله و k مجهول می‌باشد. (مجهول‌ها t_1 تا t_k هستند.) با توجه به تشکیل دترمینان ضرایب. این نتیجه به‌دست می‌آید که اگر x_i ها متمایز باشند این دستگاه معادلات یک جواب منحصربه‌فرد دارد، یعنی با دادن k عنصر اول دنباله، می‌توان جوابی منحصربه‌فرد برای دنباله پیدا کرد. (تحقیق کنید چرا در صورت متمایز بودن x_i ها، این دستگاه معادلات یک جواب منحصربه‌فرد دارد! برای این کار باید دترمینان ماتریس ضرایب را تشکیل دهیم. می‌توانید به مرجع [?] مراجعه کنید.)

مثال ۲. دنباله‌ی فیبوناچی که در مثال (۱) تعریف شده است، را حل کنید.

حل: معادله‌ی سرشت‌نمای این رابطه به صورت $x^2 - x - 1 = 0$ است و ریشه‌های آن $x_1 = \frac{1+\sqrt{5}}{2}$ و $x_2 = \frac{1-\sqrt{5}}{2}$ است. پس

[10] characteristic equation

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$f_n = t_1\left(\frac{1+\sqrt{5}}{2}\right)^n + t_2\left(\frac{1-\sqrt{5}}{2}\right)^n$$

و با توجه به مقادیر اولیه داریم:

$$\begin{cases} t_1 + t_2 = f_0 = 0 \\ t_1\left(\frac{1+\sqrt{5}}{2}\right) + t_2\left(\frac{1-\sqrt{5}}{2}\right) = f_1 = 1 \end{cases}$$

و از این معادله‌ها نتیجه میشود:

$$t_1 = \frac{1}{\sqrt{5}}, t_2 = -\frac{1}{\sqrt{5}}$$

$$f_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \qquad (5)$$

قابل توجه است، جمله‌ی $\left(\frac{1-\sqrt{5}}{2}\right)^n$ بابزرگتر شدن n بسیار کوچک می‌شود و با توجه به اینکه f_n عددی حسابی است، اگر $\langle x\rangle$ را نزدیکترین عدد صحیح به x تعریف کنیم، داریم:

$$\langle x\rangle = \lfloor x + \frac{1}{2}\rfloor$$

و با این تعریف:

$$f_n = \left\langle \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n \right\rangle \qquad (6)$$

در یافتن جوابهای دستگاه به دست آمده برای یافتن ضرایب، این شرط را قرار دادیم که x_iها متمایز باشند. حال اگر x_i ریشه‌ی مضاعف درجه‌ی ۲ معادله‌ی سرشت‌نما باشد، به راحتی قابل تحقیق است که $a_n = nx_i^n$ نیز یک جواب رابطه‌ی بازگشتی است (اثبات با مشتق‌گیری از معادله‌ی سرشت‌نما است، به این وسیله که ریشه‌ی مضاعف، ریشه‌ی مشتق معادله‌ی سرشت‌نما است.) به همین طریق می‌توان استدلال کرد که اگر x_i ریشه‌ی مضاعف درجه ۳ باشد، $n^2x_i^n$ نیز یک جواب رابطه‌ی بازگشتی است. در حالت کلی اگر x_i ریشه‌ی مضاعف درجه p باشد،

$$g(n) = t_0x^n + t_1nx^n + t_2n^2x^n + \cdots + t_{p-1}n^{p-1}x^n$$

جوابی برای رابطه‌ی بازگشتی است.

مثال ۳. رابطه‌ی بازگشتی زیر را حل کنید:

$$a_n = -a_{n-1} + 3a_{n-2} + 5a_{n-3} + 2a_{n-4} (n \geq 5)$$

حل: ریشه‌های معادله سرشت‌نمای آن به صورت زیر درآید:

$$x^4 + x^3 - 3x^2 - 5x - 2 = 0 \Rightarrow a_n = t_1(-1)^n + t_2n^2(-1)^n + t_4 2^n$$

و با معلوم بودن a_1 تا a_4 مقادیر t_1 تا t_4 به دست می‌آیند. مثلاً در حالت $a_1 = 4$ و $a_2 = -3$ و $a_3 = 22$ و $a_4 = -7$ از دستگاههای متناظر جوابهای $t_1 = -t_3 = 1$ و $t_4 = -t_2 = 2$ به دست می‌آیند.

مثال بعد، حالتی را بررسی می‌کند که در آن معادله‌ی سرشت‌نما ریشه‌ی حقیقی ندارد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مثال ۴. رابطه‌ی بازگشتی $a_n = 2a_{n-1} - 2a_{n-2}$ $(a_0 = 1, a_1 = 0)$ را حل کنید.

حل: معادله‌ی سرشت‌نمای آن به صورت $x^2 - 2x + 2 = 0$ است که جواب غیر حقیقی مقابل را دارد. $(\alpha = 1 + i, \bar{\alpha} = 1 - i)$، پس داریم:

$$\begin{cases} \alpha = \sqrt{2}\ (\cos(\frac{\pi}{4}) + i\sin(\frac{\pi}{4})) \\ \bar{\alpha} = \sqrt{2}\ (\cos(\frac{\pi}{4}) - i\sin(\frac{\pi}{4})) \end{cases}$$

$$\Rightarrow a_n = A\alpha^n + B\bar{\alpha}^n = \left\{(\sqrt{2})^n \left[A\left(\cos(\frac{n\pi}{4}) + i\sin(\frac{n\pi}{4})\right)\right] + \left[B\left(\cos(\frac{n\pi}{4}) - i\sin(\frac{n\pi}{4})\right)\right]\right\}$$

$$\Rightarrow a_n = \sqrt{2}\left(C\cos(\frac{n\pi}{4}) + D\sin(\frac{n\pi}{4})\right)$$

$$a_0 = 1, a_1 = 0 \Rightarrow C = 1, D = -1$$

$$\Rightarrow a_n = \sqrt{2}\left((\cos(\frac{n\pi}{4}) - \sin(\frac{n\pi}{4})\right)$$

لازم به تذکر است در این معادلات از رابطه‌ی مهم زیر استفاده شده است که با استفاده از استقرا ثابت می‌شود.

$$(\cos\theta + i\sin\theta)^n = \cos(n\theta) + i\sin(n\theta)$$

۵.۷.۱ رابطه‌ها بازگشتی غیر همگن با ضرایب ثابت:

رابطه‌ی بازگشتی درجه‌ی k زیر غیر همگن است:

$$a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k} + f(n)$$

اگر A_n در رابطه‌ی بازگشتی همگن ۱ صدق کند و B_n در رابطه‌ی بازگشتی غیرهمگن صدق کند آنگاه $A_n + B_n$ نیز در رابطه‌ی غیرهمگن صدق می‌کند. زیرا:

$$c_1(A_{n-1} + B_{n-1}) + \cdots + c_k(A_{n-k} + B_{n-k}) + f(n) =$$

$$(c_1 A_{n-1} + \cdots + c_k A_{n-k}) + (c_1 B_{n-1} + \cdots + c_k B_{n-k} + f(n)) = A_n + B_n$$

در این حالت A_n را جواب قسمت همگن رابطه و B_n را یک جواب خاص آن گویند. مثلاً اگر $a_n = 3a_{n-1} + 2 - 2n^2$، آنگاه $A_n = t_1 3^n$ و چون $f_n = 2 - 2n^2$، $B_n = pn^2 + qn + r$ می‌گیریم، حال داریم:

$$pn^2 + qn + r = 3(p(n-1)^2 + q(n-1) + r) + 2 - 2n^2$$

در نتیجه، برای این که این رابطه یک اتحاد برای n باشد، داریم: $D = 2, C = 3, B = 1$. در این صورت، $B_n = n^2 + 3n + 2$ و خواهیم داشت:

$$a_n = A_n + B_n = t_1 3^n + n^2 + 2$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

با فرض $n_0 = 3$ داریم: $t_1 = 1$ پس:

$$a_n = 3^n + n^2 + 3n + 2$$

۶.۷.۱ حل رابطه‌ها بازگشتی غیر همگن:

اگر رابطه‌ی بازگشتی غیر همگن را به صورت زیر داشته باشیم:

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + b^n P(n) \qquad (7)$$

که در آن b ثابت است و $P(n)$ چند جمله‌ای، از درجه‌ی d بر حسب n باشد، برای این رابطه، مشابه رابطه‌ها بازگشتی همگن معادله‌ی سرشت‌نما به صورت زیر تعریف می‌شود:

$$(x^k - c_1 x^{k-1} - c_2 x^{k-2} - \cdots - c^k)(x - b)^{d+1} = 0 \qquad (8)$$

با داشتن این معادله و به دست آوردن ریشه‌های (x_i) آن مشابه قبل، جوابهای این معادله به صورت ترکیب خطی x_i^n بیان می‌شود. (تحقیق این موضوع به عهده‌ی شما.) به طور کلی ریشه‌های معادله‌ی سرشت‌نمای یک رابطه‌ی بازگشتی مشخص‌کننده‌ی جوابهای رابطه‌ی بازگشتی است. با داشتن این معادله‌ی سرشت نما برای رابطه‌ها بازگشتی بسیاری از این رابطه‌ها باروشی مشابه رابطه‌ها بازگشتی همگن، به راحتی قابل حل است.

مثال ۵.

- الف) رابطه‌ی بازگشتی $a_n = 2a_{n-1} + (n+5)3^n$ را حل کنید.
- ب) رابطه‌ی بازگشتی $a_n = 2a_{n-1} + n$ را حل کنید.

حل:

- الف) معادله‌ی سرشت نمای این رابطه به صورت $(x-2)(x-3)^2 = 0$ در می آید، پس: $a_n = t_1 2^n + t_2 3^n + t_3 n 3^n$ که با توجه به حالتهای اولیه داده شده، میتوان t_i ها رابه دست آورد.
- ب) معادله‌ی سرشت نمای آن $(x-2)(x-1)^2 = 0$ است، پس $a_n = t_1 2^n + t_2 + t_3 n$ است.

در حالت کلی معادله‌ی سرشت‌نمای رابطه‌ی بازگشتی مقابل

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \cdots + c_k a_{n-k} + b_1^n P_1(n) + b_2^n P_2(n) + \cdots \qquad (9)$$

به صورت،

$$(x^k - c_1 x^{k-1} - c_2 x^{k-2} - \cdots - c_k)(x - b_1)^{d_1+1}(x - b_2)^{d_2+1} \cdots = 0 \qquad (10)$$

است.

مثال ۶. رابطه‌ی بازگشتی $a_n = 2a_{n-1} + n + 2^n$ را حل کنید.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

حل: معادله سرشت نمای آن به صورت $(x-2)(x-1)^2(x-2)=0$ است. پس

$$a_n = t_1 + t_2 n + t_3 2^n + t_4 n 2^n$$

و مثلاً اگر $a_0 = 0$، داریم: $a_n = -2 - n + 2^{n-1} + n2^n$.

بدین ترتیب بسیاری از رابطه‌ها بازگشتی باضرایب ثابت حل می‌شوند.

حال با حل مسئله ای کاربردی به روشی دیگر در حل رابطه‌ها بازگشتی توجه می کنیم:

مثال ۷. مسئله (بیست و یکمین المپیاد جهانی ریاضی): A و E را دو رأس روبه‌روی یک ۸ ضلعی منتظم می گیریم، قورباغه‌ای از رأس A آغاز به جهیدن می کند و هر بار به رأس مجاور می‌پرد. ولی وقتی به رأس E رسید، همانجا متوقف می‌شود. a_n را تعداد مسیرهایی می گیریم که قورباغه با n جهش از طریق آنها از A به E برسد. ثابت کنید:

$$a_{2n} = \frac{1}{\sqrt{2}}(x^{n-1} - y^{n-1})$$

که در آن $x = 2 + \sqrt{2}$ و $y = 2 - \sqrt{2}$

اثبات: ۸ ضلعی روبرو را در نظر می‌گیریم، اگر b_n تعداد مسیرهایی باشد که قورباغه از آنها با n جهش از A به E برسد. اگر قورباغه بخواهد از A به E برود: در دو جهش اول یا به C می‌رسد یا به G می‌رسد یا به A. برمی‌گردد که به دو طریق می‌تواند به A بازگردد $[ABA, AHA]$. حال از جایی که الان به آن رسیده باید شروع کند و با $n-2$ جهش به E برسد، بنابراین داریم: $a_n = 2b_{n-2} + 2a_{n-2}$. در مورد b_n نیز مشابهاً می‌توان گفت: $b_n = 2b_{n-2} + a_{n-2}$ (چرا ؟)

نخست دقت می‌کنیم چون بین A و E، ۴ ضلع وجود دارد، و جهشهای رفت و برگشت با هم تعداد زوجی می‌سازند، برای رفتن از A به E حتماً تعداد زوجی حرکت لازم است. پس $a_{2n-1} = 0$. برای حالتهای زوج دو رابطه‌ی بازگشتی $a_n = 2b_{n-2} + 2a_{n-2}$ و $b_n = 2b_{n-2} + a_{n-2}$ را داریم. حال دو روش وجود دارد، روش اول: از تفاضل دو رابطه بدست می‌آید: $b_{n-2} = a_{n-2} - a_{n-4}$ و در نتیجه با گذاشتن در رابطه‌ی اولی داریم: $a_n = 4a_{n-2} - 2a_{n-4}$ حال اگر $c_n = a_{2n}$ در نظر بگیریم، رابطه‌ی همگن $(n > 2)\ c_n = 4c_{n-1} - 2c_{n-2}$ بدست می‌آید که با توجه به حالتهای اولیه‌ی $c_1 = 0$ و $c_2 = 1$ خواهیم داشت:

$$a_{2n} = c_n = \frac{1}{\sqrt{2}}((2+\sqrt{2})^{n-1} - (2-\sqrt{2})^{n-1})$$

(با تشکیل معادله‌ی سرشت نما و حل آن بدست می‌آید) حال مشابه مثال ۵، به دلیل اینکه $2 - \sqrt{2} < 1$، می‌توان تحقیق کرد:

$$f_n = \left(\frac{(2+\sqrt{2})^{n-1}}{\sqrt{2}}\right) \qquad (11)$$

روش دوم: این روش با آنچه تا به حال گفتیم متفاوت است. ما به این روش در حل این مسئله بسنده می‌کنیم:

$$\begin{aligned} a_n &= 2a_{n-2} + 2b_{n-2} \\ b_n &= a_{n-2} + 2b_{n-2} \end{aligned}$$

حال اگر بردار V_n را به صورت $V_n = \begin{pmatrix} a_n \\ b_n \end{pmatrix}$ تعریف کنیم، باید داشته باشیم: $T = \begin{pmatrix} 2 & 2 \\ 1 & 2 \end{pmatrix}$،

$V_1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

برای تعیین بردار V_m با این حالت خاصیت مقادیر ویژه‌ی ماتریس T را تعیین می‌کنیم : این مقادیر ریشه‌های معادله‌ی مفسر ماتریس می‌باشند:

$$\begin{vmatrix} 2-\lambda & 2 \\ 1 & 2-\lambda \end{vmatrix} = \lambda^2 - 4\lambda + 2 = 0$$

و بنابراین: $\lambda_1 = 2 + \sqrt{2}$ و $\lambda_2 = 2 - \sqrt{2}$. بردارهای ویژه‌ی ماتریس T، یعنی u_1 و u_2 دارای این ویژگی هستند که: $Tu_i = \lambda_i u_i$ و $T[Tu_i] = \lambda_i^2 u_i$ و ..... که برای $i = 1, 2$، می‌توان آنها را پیدا کرد:

$$u_1 = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ \frac{1}{\sqrt{2}} \end{pmatrix}, \quad u_2 = \frac{1}{\sqrt{2}}\begin{pmatrix} -1 \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

بنابراین V_1 یک ترکیب خطی از u_1 و u_2 است. یعنی:

$$V_1 = \lambda_1 u_1 + \lambda_2 u_2 \Rightarrow V_m = T^{(m-1)} V_1 = \lambda_1^{m-1} u_1 + \lambda_2^{m-1} u_2 = \begin{pmatrix} a_{2m} \\ b_{2m} \end{pmatrix}$$

و بدین ترتیب،

$$\Rightarrow a_{2m} = \frac{1}{\sqrt{2}}[(2+\sqrt{2})^{m-1} + (2-\sqrt{2})^{m-1}] \qquad (12)$$

تمرین‌ها

۱. روی محیط دایره‌ای چند کارت کوچک سیاه و سفید قرار داده‌ایم. دو نفر به نوبت این عمل را انجام می‌دهند، اولی همه‌ی کارتهای سیاهی که در مجاورت یک سفید باشند، بر می‌دارد و دومی روی کارتهای سفید مجاور سیاه همین کار را انجام می‌دهد. اگر ۴۰ کارت وجود داشته باشد، آیا ممکن است پس از انجام ۲ حرکت فقط یک کارت باقی بماند ؟ اگر کارتها ۱۰۰۰ تا باشد، حداقل چند حرکت لازم است ؟ اگر n کارت روی دایره باشد، حداقل چند حرکت لازم است ؟

۲. تعداد زیردرختهای فراگیر گراف‌های زیر چند تاست ؟[11]

- نردبان n تایی که از دو مسیر n تایی که با یک تطابق به هم وصل شده‌اند، تشکیل شده است $(K_2 \times K_n)$
- دور n تایی که از یک دور n تایی و یک رأس که به همه‌ی رئوس دور متصل است، تشکیل شده است.

[11] زیردرخت فراگیر یک گراف زیرگراف همبندی از گراف است که دور نداشته باشد و شامل همه‌ی رئوس باشد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

فصل ۲

مرتب‌سازی و مرتبه‌ی آماری

There is nothing more difficult to take in hand, more perilous to conduct, or more uncertain in its success, than to take the lead in the introduction a new order of things. ..... (Niccolo Machiavelli, The prince, 1513)

۱.۲ الگوریتم‌های مرتب‌ساز

مساله‌ی مرتب سازی: n عنصر با کلیدهای $a_1, a_2, \ldots, a_n$ و رابطه‌ی ترتیب کامل[1] $\leq$ داده شده‌اند جای‌گشت π از عناصر داده شده را پیدا کنید به‌طوری‌که:

$$a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$$

الگوریتم‌های مرتب‌ساز[2] را از سه جنبه می‌توان بررسی کرد:

۱. از نظر موقعیت داده‌ها در زمان مرتب کردن

- الگوریتم‌های مرتب‌ساز داخلی[3]، که در آن‌ها همه‌ی داده‌ها هنگام مرتب شدن در حافظه هستند. بنابراین دست‌یابی به داده‌ها سریع صورت می‌گیرد.
- الگوریتم‌های مرتب‌ساز خارجی[4]، که در آن‌ها همه‌ی داده‌ها هنگام مرتب شدن به طور هم‌زمان در حافظه نیستند. بنابراین دست‌یابی به آن‌ها کند است. دراین الگوریتم‌ها تعداد دست‌یابی به حافظه‌ی جانبی، به دلیل کندی این عمل، اهمیت زیادی دارد بخصوص اگر داده‌ها ترتیبی[5] باشند.

۲. از نظر حفظ ترتیب نسبی عناصر

---

[1] total order. یک رابطه ترتیب جزئی (partial order) است اگر غیرمتقارن، بازتابی و متعدی باشد. رابطه‌ی ترتیب جزئی R بر روی مجموعه‌ی A رابطه‌ی ترتیب کامل است، اگر برای هر $a, b \in A$ داشته باشیم aRb یا bRa

[2] sorting algorithms

[3] internal sorting algorithms

[4] external sorting algorithms

[5] sequential

- الگوریتم‌های متعادل[۶]، که در آن‌ها ترتیب نسبی عناصر با کلیدهای یکسان قبل و بعد از عمل مرتب‌سازی یکی باشد.
- الگوریتم‌های غیرمتعادل[۷]، که در آن‌ها ترتیب نسبی عناصر مساوی در انتها لزوماً حفظ نشود.

۳. از نظر نحوه‌ی مرتب کردن داده‌ها

- الگوریتم‌های مبتنی بر مقایسه[۸]، که بر اساس مقایسه کلیدهای عناصر عمل می‌کنند. این الگوریتم‌ها اطلاعات دیگری از داده‌ی ورودی ندارند و فقط با انجام مقایسه بین کلیدهای عناصر می‌توانند ترتیب نسبی آن‌ها را پیدا کنند.
- الگوریتم‌هایی که بر اساس مقایسه‌ی عناصر عمل نمی‌کنند[۹]. این الگوریتم‌ها با استفاده از اطلاعاتی که از قبل در باره‌ی نوع کلیدها موجود است، و در شرایط خاص، از روش‌هایی استفاده می‌کنند که در آن‌ها کلیدهای عناصر با هم مقایسه نمی‌شوند.

بدیهی است که حد پایین برای هر الگوریتم مرتب‌ساز، از جمله الگوریتم‌هایی که مبتنی بر مقایسه نیستند $\Omega(n)$ است. برای الگوریتم‌های مبتنی بر مقایسه حد پایین هم در بدترین حالت و حالت متوسط $\Omega(n \log n)$ است.

ابتدا به بررسی الگوریتم‌هایی که بر اساس مقایسه عناصر عمل نمی‌کنند می‌پردازیم که از آن‌ها Count Sort، Radix Sort و Bucket Sort را می‌توان نام برد. دراین الگوریتم‌ها کلید شامل تعدادی مولفه (مستقل از n تعداد داده‌ها) می‌باشد و هر مولفه حوزه‌ی محدود و قابل شمارش دارد. قبل از ارایه‌ی الگوریتم، به یک تعریف توجه کنید.

تعریف: ترتیب الفبایی[۱۰]. اگر $\vec{b}$ و $\vec{a}$ دو رشته به صورت زیر باشند:

$$\vec{a} = a_1 a_2 \ldots a_i \ldots a_n$$
$$\vec{b} = b_1 b_2 \ldots b_i \ldots b_m$$

می‌گوییم $\vec{a} \leq \vec{b}$ اگر برای $i < k \leq n$، داشته باشیم $a_i = b_i$ و $a_k < b_k$. یا $n < m$ و برای $1 \leq i \leq n$ داشته باشیم $a_i = b_i$. عمل مرتب‌سازی در مواردی که کلیدها رشته باشند بر اساس ترتیب الفبایی است.

۱.۱.۲ درخت تصمیم

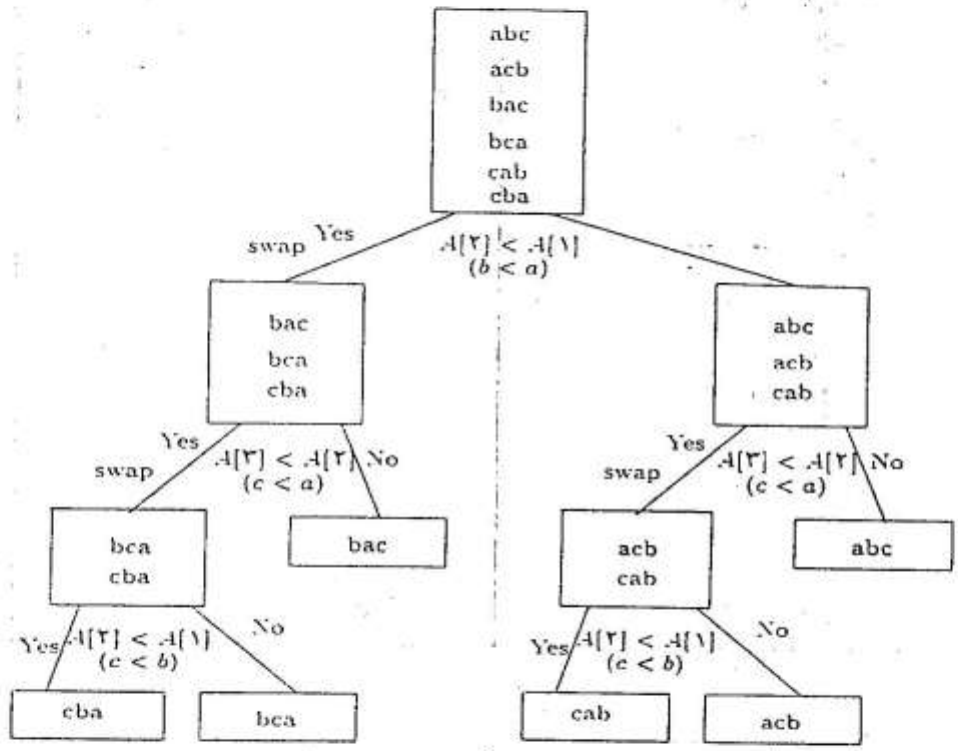
یک الگوریتم که بر روی عناصر ورودی عمل مقایسه انجام می‌دهد را می‌توان با «درخت تصمیم[۱۱]» مدل کرد. به‌طور مثال، الگوریتم مرتب‌سازی «مبتنی بر درج» یا Insertion Sort روی سه عنصر a، b، و c به صورت شکل ۱.۲ مدل شده است. بدیهی است که هر درخت تصمیم یک الگوریتم مرتب‌ساز یک درخت دودویی کامل است که برگ‌های آن جای‌گشت‌های مختلف ورودی هستند.

قضیه ۲. یک درخت دودویی با ارتفاع h حداکثر 2^h برگ دارد.

---

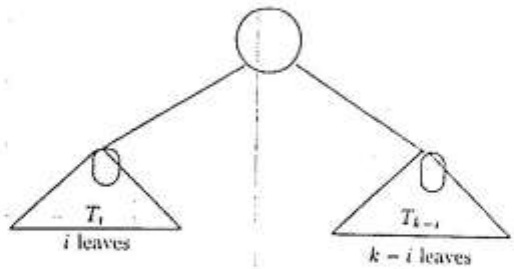
[۶] stable
[۷] unstable
[۸] comparison sort
[۹] non-comparison sort
[۱۰] Lexicographic Ordering
[۱۱] decision tree

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱.۲: درخت تصمیم Insertion Sort

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲.۲: اثبات قضیه

این قضیه با استقرا روی h اثبات می‌شود (امتحان کنید).

قضیه ۳. ارتفاع یک درخت تصمیم که n عنصر را مرتب می‌کند حداقل $\lceil \log n! \rceil$ است.

اثبات. این درخت تصمیم حداقل $n!$ برگ دارد، بنابراین ارتفاعش حداقل $\lceil \log n! \rceil$ است. □

نتیجه: هر الگوریتم مرتب‌ساز مبتنی بر مقایسه، برای مرتب کردن n عنصر در بدترین حالت حداقل $\lceil \lg n! \rceil$ مقایسه انجام می‌دهد.

می‌دانیم که $n! \le n^n$، پس $\lceil \lg n! \rceil \le n \lg n$. ولی این حد بالای ضعیفی است. تقریب استرلینگ[۱۲] حد به‌تری را به‌دست می‌دهد. براساس این تقریب داریم:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

با استفاده از این فرمول می‌توان اثبات کرد که:

$$\begin{aligned} n! &= o(n^n) \\ n! &= \omega(2^n) \\ \lg(n!) &= \Theta(n \lg n) \end{aligned}$$

قضیه ۴. اگر کلیه‌ی جای‌گشت‌های یک ترتیب nتایی با احتمال یکسان در ورودی ظاهر شوند، آن گاه متوسط عمق برگ‌های این درخت حداقل $\log n!$ خواهد بود.

اثبات. فرض می‌کنیم $D(T)$ مجموع عمق برگ‌های یک درخت دودویی T و $D(m)$ کوچک‌ترین مقدار $D(T)$ برای کلیه‌ی درخت‌های دودویی T با m برگ باشد. با استقرا ثابت می‌کنیم که $D(m) \ge m \log m$.

۱. پایه: $m = 1$ واضح است.

[۱۲] Stirling's Approximation

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۲. فرض: برای $m < k$ درست است.

۳. حکم: T با k برگ را در نظر بگیرید (شکل ۲.۲).

$$
\begin{aligned}
D(T) &= i + D(T_i) + (k - i) + D(T_{k-i}) \\
D(k) &= \min_{1 \le i \le k} \{k + D(i) + D(k-i)\} \\
D(k) &\ge k + \min_{1 \le i \le k} \{i \log i + (k-i) \log (k-i)\}
\end{aligned}
$$

با استقرا ثابت می‌شود که برای اعداد طبیعی مقدار کمینه در $i = \frac{k}{2}$ اتفاق می‌افتد. پس $D(k) \ge k \log k$ و $D(m) \ge m \log m$.

□

۲.۲ الگوریتم‌های مرتب‌ساز خطی

۱.۲.۲ الگوریتم Count Sort

ورودی n عنصر با کلیدهای بین ۱ تا m می‌باشد. می‌خواهیم این عناصر را جابه‌جا بکنیم به‌طوری‌که بر اساس کلیدهایشان مرتب شوند. در این الگوریتم ورودی در آرایه‌ی A قرار دارد و خروجی در آرایه‌ی B ذخیره می‌شود. الگوریتم ابتدا تعداد عناصری را که کلیدشان برابر یک مقدار کلید، مثلاً r است معین می‌کند و از روی آن بزرگترین اندیس در آرایه‌ی B برای عناصر با کلید r را محاسبه می‌کند. در انتها با یک بار بررسی آرایه‌ی A، هر عنصر آن در جای درست در آرایه‌ی B قرار می‌گیرد.

```
Count_Sort(A,B,m)
    for i<-1 to m
        do  c[i] <- 0
    for i <-1 to length(A)
        do C[A[i]] <- C[A[i]]+1
    for i <-2 to m
        do C[i] <- C[i]+C[i-1]
    for i <- length(A) downto 1
        do B[C[A[i]]] <- A[i]
           C[A[i]] <- C[A[i]]-1
```

یک عنصر در جای مناسب قرار می‌گیرد و چون در هر مرحله عناصر قبلی که در جای خود قرار گرفته‌اند، دست‌کاری نمی‌شوند بعد از n مرحله همه‌ی عناصر مرتب می‌شوند. واضح است که زمان اجرا $O(n)$ می‌باشد.

مقدار C[i] از آرایه‌ی C[1..m] در انتهای گام چهارم تعداد عناصر با کلید i را نشان می‌دهد و در انتهای گام ششم این درایه تعداد عناصری که کلید آن‌ها حداکثر i است را ذخیره می‌کند. در هر بار حلقه‌ی آخر، یک عنصر از

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

A در جای نهایی خود در آرایه‌ی B قرار می‌گیرد. دلیل آن‌که این حلقه از انتها به ابتدای A نکرار می‌شود آن است که الگوریتم متعادل شود. زمان اجرای کل الگوریتم $O(n+m)$ می‌باشد که اگر m از $O(n)$ باشد زمان اجرای کل $O(n)$ خواهد بود.

اگر کلیدهای عناصر اعداد ۱ تا n باشند (از هر کلید یک عدد)، می‌توان با الگوریتم زیر آن‌ها را به‌صورت «درجا» (بدون استفاده از آرایه‌ی کمکی) مرتب کرد.

```
Count_Sort(A,n)
    for i<-1 to n
        do  while A[i].key <> i
                do  Swap(A[i],A[A[i].key])
```

در این الگوریتم با هر بار تعویض، حداقل یک عنصر در جای نهایی خودش قرار می‌گیرد (چرا؟) هم‌چنین اگر از هر کلید بیش‌از یک عدد داشته باشیم الگوریتم ممکن است در حلقه بیفتد.

۲.۲.۲ الگوریتم Radix Sort

این یک الگوریتم قدیمی است که در ماشین‌های مرتب‌ساز کارت‌ها از آن استفاده می‌شد. در این الگوریتم (اگر برای مرتب کردن اعداد به کار گرفته شود) ورودی‌ها را ابتدا بر اساس کم‌ارزش‌ترین رقم و سپس دومین کم‌ارزش‌ترین رقم و به همین ترتیب مرتب می‌کنیم. برای این کار باید از یک الگوریتم مرتب‌ساز متعادل مانند Count_Sort استفاده کرد.

```
Radix_Sort(A, d)
    for i<-1 to d
        do  sort array A on digit i by a stable sort
```

d تعداد رقم‌های ورودی می‌باشد. اگر تعداد رقم‌های ورودی یکسان نباشند با گذاشتن صفر در سمت چپ آن‌ها تعداد رقم‌ها را مساوی می‌کنیم.

درستی الگوریتم را می‌توان با استقرا بر روی i اثبات نمود. به این صورت که در انتهای مرحله‌ی i ام، عناصر برحسب i رقم آخرشان مرتب می‌شوند. پایه‌ی استقرا برای $i = 0$ روشن است. اگر این امر برای $i = k$ درست باشد، با توجه به این که از یک الگوریتم متعادل استفاده می‌کنیم، در انتهای مرحله‌ی $k+1$ ام نیز ادعا درست است.

زمان اجرای Radix Sort به الگوریتم ثانویه به کار رفته نیز بستگی دارد. اگر از Count Sort استفاده کنیم زمان اجرا از $O(dn)$ خواهد بود.

Radix Sort را می‌توان تعمیم داد برای حالتی که ورودی آرایه‌ای از رکوردها باشد که هر رکورد دارای کلیدی با k مؤلفه به نام‌های $f_1 \ldots f_k$ از داده‌گونه‌های به ترتیب $t_1 \ldots t_k$ باشد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۳.۲.۲ الگوریتم Bucket Sort

در این الگوریتم ورودی می‌تواند لیست A با n رکورد، که هر رکورد دارای کلیدی با k مؤلفه به نام‌های $f_1 \ldots f_k$ از داده‌گونه‌های به ترتیب $t_1 \ldots t_k$ می‌باشد، باشد. تعداد حالت‌های t_i را s_i فرض می‌کنیم. در مرتب کردن، f_1 بر f_{k-1} و f_k اولویت دارد.

برای $(1 \le i \le k)$ داریم: B_i : array[t_i] of list-type.

```
Bucket_Sort
    for i<-k downto 1
        do for each value v of type t_i
                do make B_i[v] empty
    for each record r in list A
        do move r from A to the end of B_i[v]
            (* where v is the value of f_i of the key for r *)
    for each value v of type t_i from lowest to highest
        do concat B_i[v] to the end of A)
```

اگر هر سطل را به‌صورت یک صف پیاده‌سازی کنیم، هر عمل concat در زمان ثابت قابل اجرا است. زمان اجرا این الگوریتم برابر است با:

$$\sum_{i=1}^{k} O(s_i + n) = O(kn + \sum_{i=1}^{k} s_i)$$

اگر $s_i = n$ آن گاه

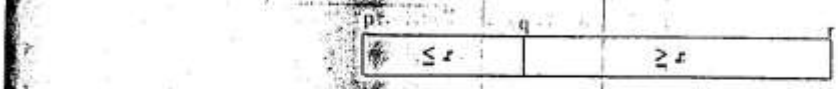
$$T(n) = \Theta(kn + \sum_{i=1}^{k} n) = \Theta(n + kn) = \Theta(n)$$

مثال: فرض کنید کلیدها شامل سه مؤلفه با مقادیر a..z، ۱۰۰ ... ۱ و ۱۴۰۰ ... ۱۳۰۰ هستند و ۷ رکورد زیر داده شده اند.

a_1	a	۵	۱۳۲۰
a_2	c	۱۲	۱۳۱۰
a_3	b	۱۲	۱۳۰۵
a_4	a	۸	۱۴۰۰
a_5	z	۱۰	۱۳۰۸
a_6	b	۱۲	۱۳۰۴
a_7	a	۶	۱۳۱۰

سه سطل (Bucket) برای کلیدها در نظر می‌گیریم. ابتدا بر اساس key[3] داده‌ها را در سطل مربوطه می‌انداریم. سپس آن‌ها را به ترتیب از سطل‌ها برداشته و صف جدید را تولید می‌کنیم و به همین ترتیب در مورد سطل‌های دیگر عمل می‌کنیم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳.۲: تقسیم‌بندی در الگوریتم quicksort

صف خروجی ۳	Bucket[1]	صف خروجی ۲	Bucket[2]	صف خروجی ۱	Bucket[3]	صف ورودی
a_1	a_1, a_7, a_2	a_1	a_1	a_6	a_6	a_1
a_7	a_6, a_3	a_7	a_7	a_2	a_3	a_2
a_2	a_2	a_2	a_2	a_5	a_5	a_3
a_6	a_5	a_5	a_5	a_3	a_2, a_7	a_4
a_3		a_6	a_6, a_3, a_2	a_7	a_1	a_5
a_4		a_3		a_1	a_4	a_6
a_5		a_4		a_4		a_7

۳.۲ الگوریتم‌های مرتب‌ساز مبتنی بر مقایسه

«تنها چیزی که تحقق یک رؤیا را غیر ممکن می‌سازد، ترس از شکست است.[۱۲]»

۱.۳.۲ الگوریتم quicksort

quicksort الگوریتمی است که یک آرایه‌ی n عنصری را در بدترین حالت با $O(n^2)$ و در حالت متوسط (چیزی که معمولاً از آن انتظار داریم) در $O(n \log n)$ مرتب می‌کند. البته ضریب ثابت $n \log n$ کاملاً کوچک است. این الگوریتم حتا برای محیط‌های حافظه مجازی نیز کارا می‌باشد.

توصیف الگوریتم

quicksort مانند merge-sort مبتنی بر روش تقسیم‌وحل است. فرض کنید می‌خواهیم آرایه‌ی A[p..r] را مرتب کنیم. این الگوریتم شامل سه مرحله‌ی زیر است.

۱. تقسیم: آرایه‌ی A[p..r] به دو آرایه‌ی ناتهی A[p..q] و A[q+1..r] تقسیم می‌شود به طوری که هر عنصر زیرآرایه‌ی A[p..q] از هر عنصر زیرآرایه‌ی A[q+1..r] بیشتر نباشد (شکل ۳.۲).

۲. حل: دو زیرآرایه‌ی A[p..q] و A[q+1..r] هرکدام به صورت بازگشتی مرتب می‌شوند.

۳. ترکیب: چون زیرآرایه‌ها به صورت درجا مرتب شده‌اند، دیگر نیازی به ترکیب نیست و آرایه مرتب شده است.

پردازه‌ی زیر quicksort را پیاده‌سازی می‌کند.

---

[۱۲] کیسابل بوتیسی، رمان الکیمیا

[۱۳] partition

[۱۴] in-place

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
QuickSort(A,p,r)
    if p<r
        then q<-partition(A,p,r)
             QuickSort(A,p,q)
             QuickSort(A,q+1,r)
```

برای این‌که تمام یک آرایه مرتب شود پرداره به‌صورت QuickSort(A,1,length(A)) فراخوانی می‌شود.

پردازه‌ی Partition که تقسیم‌بندی را انجام می‌دهد، پیاده‌سازی‌های مختلف می‌تواند داشته باشد؛ یکی از آن‌ها چنین است:

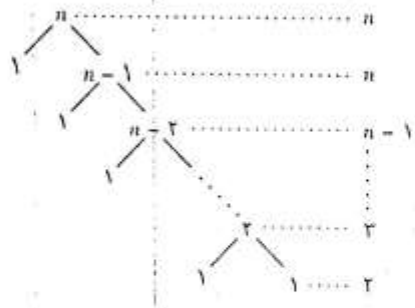
```
Partition(A,p,r)
    x ← A[p]
    i ← p-1
    j ← r+1
    while TRUE
        do repeat j ← j-1
           until A[j] <= x
           repeat i ← i+1
           until A[i] >= x
           if i<j
              then exchange A[i] <-> A[j]
              else return j
```

در این پیاده‌سازی، Partition چنین عمل می‌کند. ابتدا عنصر اول به‌نام x را به عنوان محور[16] انتخاب می‌کند (در این جا A[p]، ولی محور می‌تواند هرکدام از عناصر p تا r-1 هم باشد؛ اگر محور A[q] باشد و برحسب اتفاق بزرگترین عنصر، این پردازه در حلقه بی‌انتها گیر خواهد کرد.)

سپس i را مساوی p - 1 و j را برابر r + 1 قرار می‌دهد (دیده می‌شود که دراولین مرحله i و j اندیس‌های زیرآرایه نمی‌باشند ولی این کار کلکی جالب برای کاهش تعداد عملیات در حلقه‌ی while می‌باشد. حال در حلقه‌ی while دو حلقه‌ی repeat داریم که یکی i را به سمت راست و دیگری j را به سمت چپ حرکت می‌دهد. بدین ترتیب که i از روی عناصر A[i] < x به‌سمت راست عبور می‌کند و اگر A[i] ≥ x باشد می‌ایستد. هم‌چنین، j از روی عناصر A[j] > x به‌سمت چپ عبور می‌کند و اگر A[j] ≤ x باشد متوقف می‌شود. بدیهی است که حتماً این دو حلقه‌ی repeat ختم می‌شوند. در این صورت، اگر i < j باشد A[i] و A[j] با هم تعویض می‌شوند و حلقه‌ی while تکرار می‌شود تا وقتی که i ≥ j. توجه کنید که در انتها، رویه‌ی partition آرایه‌ی A[p..r] را به دو قسمت ناتهی تقسیم می‌کند و هیچ‌یک از عناصر قسمت اول از هیچ‌یک از عناصر قسمت دوم بزرگتر نیست. به این دلیل الگوریتم quicksort حتماً به‌درستی ختم می‌شود.

[16] pivot

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴.۲: درخت بازگشت برای $T(n) = T(n-1) + \Theta(n)$.

تحلیل الگوریتم

این الگوریتم بدترین حالت از $\Theta(n^2)$ و در حالت متوسط از $\Theta(n \log n)$ می‌باشد و این کارایی وابسته به نحوه‌ی تقسیم‌بندی است. هزینه‌ی عمل تقسیم‌بندی یک آرایه‌ی به اندازه‌ی n برابر $O(n)$ با ثابت کوچک است. چرا که در الگوریتم ارایه شده هیچ‌گاه متغیرهای i و j به‌عقب بر نمی‌گردند و تعداد تعویض‌ها حداکثر برابر $n/2$ است.

تقسیم‌بندی در بدترین حالت

بدترین رفتار quicksort هنگامی اتفاق می‌افتد که پردازه‌ی Partition در تمام مراحل الگوریتم n عنصر را به $n-1$ و ۱ عنصر تقسیم کند دراین حالت پیچیدگی زمان الگوریتم به صورت $T(n) = T(n-1) + \Theta(n)$ خواهد بود که با حل آن به‌سادگی دیده می‌شود که $T(n) = \Theta(n^2)$ (شکل ۴.۲).

توجه کنید که در این پیاده‌سازی اگر آرایه از قبل مرتب باشد باز هزینه در بدترین حالت $\Theta(n^2)$ می‌باشد (ما x را برابر A[p] قرار می‌دهیم.)

تقسیم‌بندی در بهترین حالت

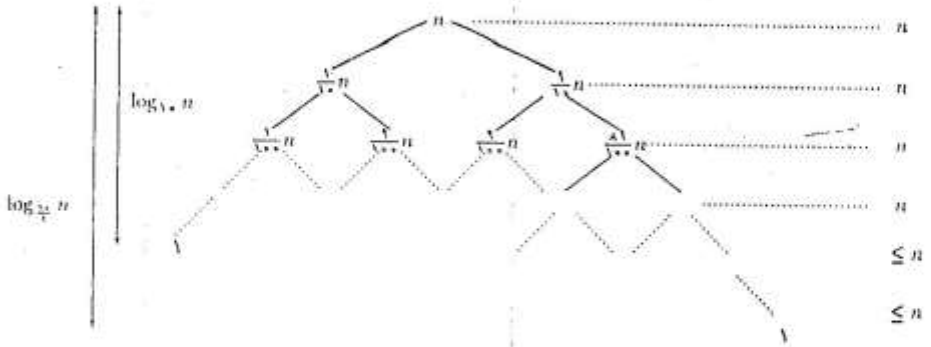
بهترین حالت هنگامی اتفاق می‌افتد که رویه‌ی Partition درهر مرحله n عنصر آرایه را به دو آرایه با تعداد عناصر $\lceil \frac{n}{2} \rceil$ و $\lfloor \frac{n}{2} \rfloor$ تقسیم کند. پیچیدگی الگوریتم در این حالت چنین محاسبه می‌شود:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n) \rightarrow T(n) = \Theta(n \log n)$$

تقسیم‌بندی متوازن

همان طور که بعداً نشان داده خواهد شد حالت متوسط به بهترین حالت نزدیک‌تر است تا به بدترین حالت. کلید فهم این مطلب در این است که توازن چگونه در رابطه بازگشتی منعکس می‌شود. برای مثال یک الگوریتم تقسیم کننده را در نظر بگیرید که همیشه آرایه را به نسبت ۹ به ۱ تقسیم می‌کند. در نگاه اول این الگوریتم کاملاً نامتوازن به نظر می‌رسد. ولی چنین نیست! رابطه‌ی بازگشتی آن $T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + n$ می‌باشد که حل آن $T(n) = O(n \log n)$. درخت بازگشت این رابطه بازگشتی در شکل ۵.۲ دیده می‌شود. توجه کنید که حتی اگر این نسبت ۹۹۹۹ به ۱ باشد (و در حالت کلی تا وقتی که همواره کسری، و نه به اندازه‌ی ثابت، از تعداد عناصر در بخش‌ها قرار گیرند) نیز الگوریتم از مرتبه‌ی $n \log n$ خواهد بود.

شهودی برای حالت متوسط



شکل ۵.۲: درخت بازگشت برای $T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + n$

انتظار ما از تابع Partition این است که هم تقسیم‌های خوب و هم بد انجام بدهد و این تقسیم‌ها به طور تصادفی در درخت بازگشت پراکنده شوند و معلوم است(!) که این تقسیم خوب و بد از تقسیم ۹ به ۱ مثال بالا بهتراست، پس این انتظار ما هم که حالت متوسط به بهترین حالت نزدیکتر باشد تا به بدترین حالت ، چندان غیر معقول نیست!
برای تحلیل الگوریتم در حالت متوسط، گونه‌ی تصادفی آن را در نظر می‌گیریم.

۲.۳.۲ گونه‌ی تصادفی quicksort

در بررسی رفتار quicksort در حالت متوسط ما فرض کردیم که همه‌ی جای‌گشت‌های عناصر ورودی با شانس یکسان ظاهر می‌شوند. اگر این فرض در مورد داده‌ی ورودی درست باشد، از نظر متوسط زمان اجرا، quicksort یک الگوریتم بهینه‌ی مرتب‌ساز است.

یک الگوریتم را «تصادفی[۱۷]» گوییم که علاوه بر آن‌که رفتار آن توسط ورودی مشخص می‌شود، رفتار آن به مقداری که مولد عدد تصادفی تولید می‌کند هم وابسته باشد. در تحلیل الگوریتم فرض می‌کنیم یک تابع Random(a,b) داریم که به صورت تصادفی آماری یک عدد بین a و b تولید می‌کند.

در گونه‌ی تصادفی quicksort، قبل از صدا زدن تابع Partition ، A[p] را با یک عدد تصادفی بین p و r عوض می‌کنیم.

```
Randomized_Partition(A,p,r)
    i <- Random(p,r)
    exchange A[p] <-> A[i]
    return Partition(A,p,r)
```

[۱۷] randomized

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Randomized_QuickSort(A,p,r)
    if p < r
       then q<-Randomized_Partition(A,p,r)
            Randomized_QuickSort(A,p,q)
            Randomized_QuickSort(A,q+1,r)
```

در بخش بعدی که کارایی الگوریتم را بررسی می‌کنیم از این الگوریتم استفاده خواهیم کرد.

تحلیل پیچیدگی زمان quicksort

در بحث قبلی گفتیم که بدترین حالت quicksort هنگامی رخ می‌دهد که تابع Partition آرایه‌ی n عنصری را به یک زیرآرایه‌ی ۱ عضوی و یک زیرآرایه‌ی $n-1$ عضوی تقسیم کند که‌در این حالت پیچیدگی الگوریتم $\Theta(n^2)$ است. این ادعا را ثابت می‌کنیم. برای این کار نشان می‌دهیم که $T(n) = O(n)$ و نیز $T(n) = \Omega(n)$.

فرض کنید $T(n)$ زمان بدترین حالت باشد و Partition آرایه‌ی n عضوی را به یک زیرآرایه‌ی q عضوی و به یک زیرآرایه‌ی $n-q$ عضوی تقسیم کند. در این صورت، $T(n) = \max\{T(q) + T(n-q)\} + \Theta(n)$. پارامتر q بین ۱ تا $n-1$ تغییر می‌کند ($1 \le q \le n-1$)، چون دو ناحیه‌ی ایجاد شده هرکدام حداقل یک عضو دارند.

با استقرا اثبات می‌کنیم که $T(n) = O(n^2)$. اگر برای $m < n$ فرض کنیم $T(m) \le cm^2$، داریم

$$T(n) \le \max\{cq^2 + c(n-q)^2\} + \Theta(n)$$

بیشینه‌ی $q^2 + (n-q)^2$ روی نواحی مرزی اتفاق می‌افتد (با رسم نمودار تابع و با درنظر گرفتن $1 \le q \le n-1$) پس در حالت $q = 1$ داریم:

$$\begin{aligned} T(n) &\le c(1 + (n-1)^2) + \Theta(n) = cn^2 - 2c(n-1) + \Theta(n) \\ T(n) &\le cn^2 \end{aligned}$$

به شرطی که c را آن‌قدر بزرگ بگیریم که $\Theta(n)$ را بپوشاند.

حال نشان می‌دهیم که $T(n) = \Omega(n^2)$.

$$T(m) \ge cm^2, m \le n \to T(n) \ge \max_{1 \le q \le n-1} \{cq^2 + c(n-q)^2\} + \Theta(n)$$

$T(m) \geq cm^2$ $\max_{1 \le q \le n-1} \{cq^2 + c(n-q)^2\} + \Theta(n)$

بنابراین،

$$T(n) \ge cn^2 - 2c(n-1) + \Theta(n) \ge cn^2$$

به‌شرطی که c آن قدر کوچک باشد که $\Theta(n) - 2c(n-1)$ مثبت شود. پس

$$T(n) = \Omega(n^2) = O(n^2) \Rightarrow T(n) = \Theta(n^2)$$

و ادعا ثابت می‌شود.

برای حالت متوسط، توجه کنید که همواره محور عنصر اول آرایه‌ی A است که به‌طور تصادفی تصادفی انتخاب می‌شود. مرتبه‌ی محور نسبت به n عنصر A می‌تواند هریک از ۱ تا n باشد. اگر احتمال این حالات را یکسان

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

بگیریم، احتمال این‌که مرتبه‌ی محور k باشد برابر $\frac{1}{n}$ است. حال توجه کنید که برای هر مرتبه‌ی $1 \le k \le n$ آرایه‌ی A به‌چه نحو تقسیم می‌شود. در هر دو حالت $k = 1$ و $k = 2$، آرایه‌ی A به دو قسمت با اندازه‌های ۱ و $n - 1$ تقسیم می‌شود. (چرا؟) بنابراین اگر $\hat{T}(n)$ متوسط زمان اجرای الگوریتم Randomized-QuickSort باشد، داریم

$$\hat{T}(n) = \frac{1}{n}\left[\hat{T}(1) + \hat{T}(n-1) + \sum_{q=1}^{n-1}[\hat{T}(q) + \hat{T}(n-q)]\right] + \Theta(n) \qquad (1)$$

در بحث‌های قبلی دیدیم که در بدترین حالت $T(n-1) = \Theta(n^2)$ است و $T(1) = \Theta(1)$ و نیز $\hat{T}(n) \le T(n)$.

پس:

$$\frac{1}{n}[\hat{T}(1) + \hat{T}(n-1)] \le \frac{1}{n}[\Theta(1) + \Theta(n^2)] = \Theta(n)$$

با توجه به فرمول ۳ داریم:

$$\hat{T}(n) = \frac{1}{n}\sum_{q=1}^{n-1}[\hat{T}(q) + \hat{T}(n-q)] + \Theta(n) = \frac{2}{n}\sum_{k=1}^{n-1}\hat{T}(k) + \Theta(n)$$

برای حل این رابطه‌ی بازگشتی، حدس می‌زنیم $\hat{T}(n) \le an\log n + b$، و تلاش می‌کنیم حدس‌مان را ثابت کنیم.

$$\begin{aligned}
\hat{T}(n) &\le an\log n + b, \quad a, b > 0 \\
\hat{T}(n) &= \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} ak\log k + b + \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} ak\log k + \frac{2}{n}\sum_{k=1}^{n-1} b + \Theta(n) \\
&= \frac{2a}{n}\sum_{k=1}^{n-1} k\log k + \frac{2b}{n}(n-1) + \Theta(n)
\end{aligned}$$

و از تقریب انتگرال ثابت می‌شود که:

$$\sum_{k=1}^{n-1} k\log k \le \frac{1}{2}n^2\log n - \frac{1}{8}n^2$$

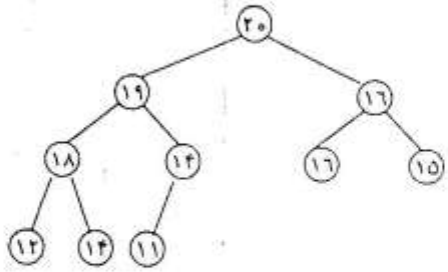
از این نامساوی و رابطه‌ی بالا داریم:

$$\begin{aligned}
\hat{T}(n) &\le \frac{2a}{n}\left(\frac{n^2}{2}\log n - \frac{n^2}{8}\right) + \frac{2b}{n}(n-1) + \Theta(n) \\
&\le an\log n + b + \left[\Theta(n) + b - \frac{an}{4}\right]
\end{aligned}$$

a و b را طوری انتخاب می‌کنیم تا درون پرانتز منفی شود. بنابراین $\hat{T}(n) \le an\log n + b$.

seyed hamed hoseinian
Software Engineering pnub

saeeditpnu.blogfa.com



۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
۲۰	۱۹	۱۶	۱۸	۱۴	۱۶	۱۵	۱۲	۱۴	۱۱

شکل ۶.۲: نمایش درختی و آرایه‌ای یک heap.

۴.۲ الگوریتم HeapSort

«تاریکترین ساعت شب، یک ساعت قبل از طلوع آفتاب است[۱۸].»

heap دودویی یک درخت دودویی کامل (به‌جز حداکثر یک گره) و متوازن (یعنی این که سطح برگ‌های آن حداکثر یک واحد اختلاف دارند) است که برگ‌های سطح آخر آن از سمت چپ چیده شده‌اند. هم‌چنین کلید هر عنصر از کلیدهای فرزندانش کوچکتر نمی‌باشد، یعنی $A[parent(i)] \geq A[i]$.

در این نمایش فرزند چپ عنصر iام در $A[2i]$ (اگر $2i \leq n$) و فرزند راست آن در $A[2i+1]$ (اگر $2i+1 \leq n$) و پدرش در $A[\lfloor \frac{i}{2} \rfloor]$ (اگر $i \neq 1$) قرار دارد.

۱.۴.۲ ویژگی‌های یک heap

لم ۱. ریشه بزرگترین عنصر heap است.

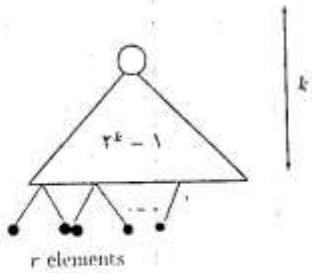
اثبات. بااستقرا بر روی ارتفاع درخت به سادگی می‌توان ثابت کرد که ریشه‌ی هر زیردرخت بزرگترین عنصر آن زیردرخت است. مبنا را یک گره‌ی تنها در نظر می‌گیریم. □

لم ۲. ارتفاع یک heap با n عنصر $h = \lfloor \log n \rfloor$ می‌باشد.

اثبات. طبق تعریف، درخت تا سطح $h-1$ پر است. هم‌چنین برای هر n یک h موجود است به طوری که $2^h \leq n < 2^{h+1}$. و یا $n = 2^h + r - 1$ که در آن $1 \leq r \leq 2^h$. heapای که با این تعداد عنصر می‌سازیم شامل

[۱۸] یک ضرب‌المثل اسپانیولی.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۷.۲: ارتفاع یک heap با n عنصر.

یک درخت دودویی پر به ارتفاع $k - 1$ و با $2^k - 1$ عنصر است که به آن r برگ در سطح k از سمت چپ اضافه شده است (شکل ۷.۲.) واضح است که ارتفاع heap برابر $\lfloor \log n \rfloor$ می‌باشد. □

اگر در تعریف بالا شرط ناکوچکتر را به نابزرگتر تبدیل کنیم، heap به صف اولویّت (priority queue) تبدیل می‌شود که در انتهای این بخش به آن اشاره خواهد شد.

درالگوریتم HeapSort از چند رویه‌ی مهم به شرح زیر استفاده شده‌است:

Heapify:

ورودی این پردازه آرایه‌ی A و اندیس مربوط به یک عنصر از آرایه است. با فرض آن که خاصیّت heap برای زیردرختان چپ و راست عنصر i برقرار است، زیردرخت به ریشه‌ی i را به صورت heap درمی‌آورد.

```
Heapify(A,i)
        l <- left(i)
        r <- right(i)
        if(l <= n) and (A[l] > A[i])
          then  largest <- l
        else  largest <- i
        if(r <=  n) and (A[r] > A[largest])
          then  largest <- r
        if(largest <> i)
          then swap(A[i], A[largest])
               Heapify(A, largest)
```

اندازه‌ی زیر درخت‌های یک heap با n عنصر حداکثر $\frac{2n}{3}$ است. این زمانی اتفاق می‌افتد که زیردرخت سمت چپ یک درخت پر به ارتفاع h و زیردرخت سمت راست یک درخت پر به ارتفاع $h - 1$ باشد. در نتیجه $T(n)$ زمان اجرای Heapify برابر است با:

$$T(n) \leq T(\frac{2n}{3}) + \theta(1)$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

و طبق قضیه‌ی اصلی داریم: $T(n) = O(\log n)$. البته بدترین حالت هنگامی روی می‌دهد که رویه تا رسیدن به یک برگ ادامه پیدا کند. بنابراین $T(n) = \Omega(\log n)$ و از این دو نتیجه می‌گیریم که $T(n) = \Theta(\log n)$.

اگر $S(n)$ حداکثر تعداد تعویض‌ها در الگوریتم فوق باشد، داریم:

$$\begin{aligned} S(n) &= \sum_{i=2}^{n} [1 + \lfloor \log i - 1 \rfloor] \\ &< n + \sum_{i=2}^{n} \log n = n + n \log n \end{aligned}$$

Build-Heap:

این پرداره یک آرایه‌ی A با n =length(A) را با استفاده از Heapify به یک Heap تبدیل می‌کند. با توجه به این که درایه‌های A[i] برای i=1... n/2 هرکدام یک heap با یک عنصر (همان A[i]) است، کار Heapify از $n/2$ تا ۱ دنبال می‌شود.

```
Build_Heap(A)
    heap_size(A) <- length(A)
    for  i <- [length(A)/2] downto 1
         Heapify(A,i)
```

حداکثر زمان اجرای این پرداره برابر است با:

$$T(n) = O(n)O(\log n) = O(n \log n)$$

اما زمان اجرای واقعی از این کمتر است.

لم ۳. حداکثر تعداد گره‌های با ارتفاع h در یک heap با n عنصر برابر است با $\lceil \frac{n}{2^{h+1}} \rceil$.

اثبات. با استقرا بر روی h. برای پایه‌ی استقرا، می‌دانیم که تعداد برگ‌های یک heap، که ارتفاع آن‌ها $h = 0$، برابر $\lceil \frac{n}{2} \rceil$ است (در نمایش آرایه‌ای، تعداد عناصر با اندیس i فرزند ندارند، یعنی $2i > n$.) فرض کنید n_h تعداد گره‌ها در ارتفاع h در درخت T با n گره باشد و نیز آن که ادعا برای گره‌های با ارتفاع $h - 1$ درست باشد. اگر T' با n' همان T باشد که برگ‌های آن را برداشته باشیم، داریم $n' = n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$. هم‌چنین می‌دانیم که گره‌هایی که ارتفاع‌شان در T برابر h است، در T' در ارتفاع $h - 1$ قرار دارند. بنابراین:

$$n_h = n'_{h-1} \le \lceil n'/2^h \rceil = \lceil \lceil n/2 \rceil / 2^h \rceil \le \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil.$$

و حکم ثابت است.

می‌دانیم که هزینه‌ی Heapify بر روی گره‌های با ارتفاع h از $O(h)$ می‌باشد. با توجه به این که $0 \le h \le \lfloor \log n \rfloor$، داریم:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h)$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

شکل ۸.۲: سمارش تعداد تعویض‌ها.

$$= O\left(\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

از آنجایی که

$$\sum_{h \geq 0} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \qquad (2)$$

بنابراین:

$$T(i) = O\left(i \sum_{h \geq 0} \frac{h}{2^h}\right) = O(n)$$

اثبات دیگر برای این نتیجه به صورت زیر است[۱۹]: اگر حداکثر تعداد تعویض‌های `Heapify(A,i)` را $S(i)$ بنامیم، با توجه به شکل ۸.۲ می‌توان دید که

$$S(i) = \begin{cases} 1 & i = \lfloor n/2^2 \rfloor + 1 \ldots \lfloor n/2 \rfloor \quad \text{تعداد} = \lfloor n/2^2 \rfloor \\ 2 & i = \lfloor n/2^3 \rfloor + 1 \ldots \lfloor n/2^2 \rfloor \quad \text{تعداد} = \lfloor n/2^3 \rfloor \\ \cdots & \cdots \\ k & i = \lfloor n/2^{k+1} \rfloor + 1 \ldots \lfloor n/2^k \rfloor \quad \text{تعداد} = \lfloor n/2^{k+1} \rfloor \\ \cdots & \cdots \\ \lfloor \log n \rfloor & i = 1 \quad \text{تعداد} = 1 \end{cases}$$

زمان اجرا متناسب است با حداکثر تعداد کل تعویض‌ها:

$$T(n) \leq \sum_{k=0}^{\lfloor \log n \rfloor} k \frac{n}{2^{k+1}}$$

که با استفاده از فرمول ۲ داریم $T(n) = O(n)$.

Heap-sort:

در این رویه ابتدا heap ساخته می‌شود. سپس ریشه که در درایه‌ی اول قرار دارد (`A[1]`) و بزرگترین عنصر آرایه است با آخرین عنصر `A[n]` تعویض می‌شود. با این کار بزرگترین عنصر در جای نهایی خود قرار می‌گیرد و از اندازه heap یکی کم می‌شود. در این مرحله پردازه‌ی `Heapify` بر روی عنصر اول آرایه و اندازه‌ی $n-1$ برای برقراری دوباره‌ی خاصیت heap اجرا می‌شود. الگوریتم به همین ترتیب n بار تکرار می‌شود تا همه‌ی عناصر مرتب شوند.

[۱۹] این اثبات در کلاس ارایه شد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
HeapSort(A)
        Build_Heap(A)
        for  i <- length(A) downto 2
             do    exchange A[1] <-> A[i]
                   heap_size(A) <- heap_size(A)-1<
                   Heapify(A,1)
```

برای محاسبه‌ی زمان اجرای الگوریتم، توجه داریم که حلقه‌ی اصلی الگوریتم $n - 1$ بار تکرار می‌شود و هر بار یک جابجایی و یک بار فراخوانی Heapify انجام می‌شود. در نتیجه:

$$T(n) = O(n) + (n - 1)O(\log n) = O(n \log n)$$

می‌توان نشان داد که زمان اجرا همیشه $\Omega(n \log n)$ می‌باشد (تحقیق کنید.)

۵.۲ میانه‌ها و مرتبه‌های آماری

مرتبه‌ی آماری[20] iام از یک مجموعه‌ی n عضوی، iامین عضو کوچک‌تر این مجموعه می‌باشد. برای مثال، کمینه[21]ی یک مجموعه مرتبه‌ی آماری یکم این مجموعه است، و بیشینه[22]ی یک مجموعه مرتبه‌ی آماری nام این مجموعه می‌باشد ($i = n$).

یک میانه به‌صورت شهودی «عضو وسطی» مجموعه می‌باشد. وقتی n فرد باشد، میانه یکتاست و مرتبه‌ی آماری $i = (n + 1)/2$ است. وقتی n زوج باشد، دو میانه داریم که مرتبه‌های آماری $i = n/2$ و $i = n/2 + 1$ هستند. بنابراین بدون در نظر گرفتن زوجیت n، میانه‌ها مرتبه‌های آماری $i = \lfloor (n + 1)/2 \rfloor$ و $i = \lceil (n + 1)/2 \rceil$ هستند.

در این بخش انتخاب مرتبه‌ی آماری iام از یک مجموعه با n عدد متفاوت را مورد بررسی قرار می‌دهیم. با این‌که برای راحتی اعداد را متفاوت در نظر گرفته‌ایم، نتایجی که در این‌جا به‌دست می‌آوریم به‌صورت شهودی قابل تعمیم به حالتی که اعداد تکراری داریم نیز هستند. مسئله‌ی انتخاب[23] را می‌توان به‌صورت زیر بیان کرد:

selection problem

ورودی : یک مجموعه A از n عدد (متفاوت) و یک عدد i که داریم $1 \le i \le n$.

خروجی : عضو $x \in A$ که دقیقاً از $i - 1$ عضو در A بزرگتر است.

مسئله‌ی انتخاب را می‌توان در $O(n \lg n)$ حل کرد، به‌این ترتیب که ابتدا اعداد را با استفاده از الگوریتم heapsort یا mergesort مرتب می‌کنیم و در این صورت عنصر iام در آرایه‌ی مرتب‌شده جواب مسئله است. ولی الگوریتم‌های سریع‌تری نیز وجود دارند.

ما ابتدا مسئله‌ی یافتن همزمان کوچک‌ترین و بزرگ‌ترین عنصر یک مجموعه را بررسی می‌کنیم. بعد به حل مسئله‌ی انتخاب می‌پردازیم. در قسمت دوم یک الگوریتم با زمان متوسط $O(n)$ را بررسی می‌کنیم و بخش سوم شامل یک الگوریتم با دیدگاهی نظری است که این مسئله را در بدترین حالت در $O(n)$ حل می‌کند.

---

[20] order statistics
[21] minimum
[22] maximum
[23] selection problem

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱.۵.۲ کمینه و بیشینه

چند مقایسه برای یافتن کمینه‌ی یک مجموعه n عضوی لازم است؟ ما به‌راحتی می‌توانیم حد بالای $n - 1$ را برای تعداد مقایسه‌ها پیدا کنیم: اعداد مجموعه را یک به یک امتحان می‌کنیم و کوچک‌ترین عضوی را که تاکنون دیده‌ایم را نگاه می‌داریم. در این روش ما فرض کرده‌ایم که مجموعه درون آرایه A ذخیره شده است، که در آن $length[A] = n$.

```
Minimum(A)
min ← A[1]
for i ← 2 to length[A]
        do if min > A[i]
                then min ← A[i]
return min
```

البته، پیداکردن بیشینه نیز با $n - 1$ مقایسه میسر است.

آیا این به‌ترین راه حل است؟ بله، برای این که ما می‌توانیم یک حد پایین از $n - 1$ مقایسه را برای یافتن کمینه به‌دست آوریم. الگوریتمی را تصور کنید که یافتن کمینه را به‌صورت یک دوره مسابقه بین عناصر در نظر می‌گیرد. هر مقایسه یک مسابقه است که در آن عضو کوچک‌تر پیروز می‌شود. نکته‌ی کلیدی این است که همه‌ی عناصر به جز برنده‌ی نهایی باید حداقل در یک مسابقه شکست خورده باشند. بنابراین، حداقل $n - 1$ مقایسه برای یافتن کمینه لازم است، و الگوریتم Minimum از نظر تعداد کل مقایسه‌ها به‌ترین است.

یک نکته‌ی جالب در تحلیل این الگوریتم، یافتن متوسط تعداد اجرا شدن خط شماره‌ی چهار است. مسئله‌یی در کتاب از شما می خواهد تا نشان دهید این مقدار $\Theta(\lg n)$ است.

یافتن هم‌زمان بیشینه و کمینه

در برخی از کاربردها، ما باید بیشینه و کمینه یک مجموعه n عضوی را باهم پیدا کنیم. برای مثال، یک برنامه‌ی گرافیکی ممکن است نیاز داشته باشد تا یک سری داده‌های به‌صورت (x, y) را طوری مقیاس کند که درون یک صفحه‌ی نمایش یا هر دستگاه گرافیکی مستطیل‌شکل دیگر قرار بگیرند. برای انجام این کار، برنامه ابتدا باید بیشینه و کمینه را در هر مختصه پیدا کند.

یافتن الگوریتمی که بتواند بیشینه و کمینه از n عنصر را به‌طور هم‌زمان در تعداد مقایسه‌هایی از مرتبه‌ی $\Omega(n)$ که از نظر آماری بهینه نیز هست پیدا کند، چندان مشکل نیست. کافی است تا بیشینه و کمینه را به‌صورت جدا از هم، و هر کدام را با $n - 1$ مقایسه پیدا کنیم؛ و در مجموع $2n - 2$ مقایسه انجام داده‌ایم.

در حقیقت، تنها $3\lceil n/2 \rceil$ مقایسه برای یافتن هم‌زمان کمینه و بیشینه لازم است. برای انجام این کار، ما عناصر بیشینه و کمینه‌ای که تا به حال دیده‌ایم را مدیریت می‌کنیم. بجای پردازش هر عضو مجموعه ورودی با روند یک مقایسه با بیشینه‌ی فعلی و یک مقایسه با کمینه‌ی فعلی، که در مجموع هزینه‌ای برابر دو مقایسه برای هر عنصر را دارد، ما عناصر را جفت جفت پردازش می‌کنیم. ما ابتدا دو عنصر ورودی را با یک‌دیگر مقایسه می‌کنیم، و سپس عنصر کوچک‌تر را با کمینه‌ی فعلی و عنصر بزرگ‌تر را با بیشینه‌ی فعلی مقایسه می‌کنیم، که با هزینه‌ی سه مقایسه به ازای هر دو عنصر انجام می‌شود.

تمرین‌ها

تمرین ۱-۱

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

نشان دهید که دومین کمینه از میان n عنصر را می‌توان در بدترین حالت با $n + \lceil \lg n \rceil - ۲$ مقایسه یافت. (راهنمایی: کمینه را هم پیدا کنید.)

تمرین ۲-۱ •

نشان دهید که در بدترین حالت، $\lceil ۳n/۲ \rceil$ مقایسه برای یافتن هم‌زمان بیشینه و کمینه از میان n عدد ضروری است. (راهنمایی: در نظر بگیرید که کدام اعداد دارای استعداد کمینه بودن و کدام دارای استعداد بیشینه بودن هستند و نشان دهید که هر مقایسه چگونه بر روی این‌ها تاثیر می‌گذارد.)

۲.۵.۲ انتخاب در زمان متوسط خطی

در وهله‌ی اول حل مسئله‌ی عمومی انتخاب[۲۴] بسیار مشکل‌تر از مسئله‌ی یافتن کوچکترین عنصر به‌نظر می‌رسد، ولی، به‌طور غیر منتظره‌ای، مرتبه‌ی زمان اجرای هردو مسئله باهم برابر است: $\Theta(n)$. در این بخش ما یک الگوریتم تقسیم‌وحل برای مسئله‌ی انتخاب ارایه می‌دهیم. الگوریتم Randomized-Select مشابه‌ی الگوریتم quicksort در فصل ۲ می‌باشد. ایده‌ی ما، همانند quicksort، تقسیم آرایه ورودی به‌صورت بازگشتی است. ولی بر خلاف quicksort، که به‌صورت بازگشتی برروی هر دو طرف نقطه‌ی تقسیم فراخوانده می‌شد، Randomized-Select تنها بر یک طرف نقطه‌ی تقسیم عمل می‌کند. این تفاوت در بررسی زمان اجرا خود را نشان می‌دهد: باوجود این که زمان اجرای متوسط quicksort از $\Theta(n \lg n)$ است، زمان اجرای متوسط Randomized-Select، $\Theta(n)$ است.

Randomized-Select از رویه‌ی Randomized-Partition که در فصل ۸.۳ کتاب معرفی شده است استفاده می‌کند. بنابراین، مشابه Randomized-Quicksort، یک الگوریتم تصادفی است، زیرا بخشی از رفتار آن با استفاده از خروجی تولیدکننده‌ی عدد تصادفی تعیین می‌شود. برنامه‌ی زیر برای Randomized-Select iامین کمینه را در آرایه‌ی $A[p..r]$ برمی‌گرداند.

```
Randomized-Select(A, p, r, i)
if p = r
    then return A[p]
q ← Randomized-Partition(A, p, r)
k ← q − p + 1
if i ≤ k
    then return Randomized-Select(A, p, q, i)
    else return Randomized-Select(A, q + 1, r, i − k)
```

```
Randomized-Select(A,p,r,i)
if (p=r)
    then return A[p]
q ← Randomized-Partition(A,p,r)
k ← q-p+1
if i≤k
    then return
```

بعد از این که Randomized-Partition در خط سوم الگوریتم اجرا می‌شود، آرایه‌ی $A[p..r]$ به دو زیرآرایه‌ی غیر تهی $A[p..q]$ و $A[q+1..r]$ تقسیم می‌شود به‌طوری‌که هر عضو $A[p..q]$ از هر عضو $A[q+1..r]$ کوچک‌تر است. خط چهارم در الگوریتم عدد k یعنی تعداد عناصر در زیرآرایه $A[p..q]$ را محاسبه می‌کند. حال الگوریتم تشخیص می‌دهد که iامین کمینه در کدام‌یک از زیرآرایه‌های $A[p..q]$ و $A[q+1..r]$ است. اگر $i \le k$، آن‌گاه عنصر دل‌خواه در قسمت پایینی نقطه‌ی تقسیم است. و در خط ششم به‌صورت بازگشتی از زیرآرایه مذکور انتخاب می‌شود. اما اگر $i > k$ باشد، آن‌گاه عنصر دلخواه در بخش بالایی نقطه‌ی تقسیم است. و چون ما هم‌اکنون k عنصر از $A[p..r]$ که کوچک‌تر

[۲۴] selection

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

از iامین کمینه هستند را می‌شناسیم—در واقع عناصر $A[p,q]$—، پس عنصر دلخواه ما $(i-k)$امین عنصر کمینه در $A[q+1,r]$ است، که به صورت بازگشتی در خط هفتم پیدا می‌شود.

بدترین زمان اجرای الگوریتم Randomized-Select از مرتبه‌ی $\Theta(n^2)$ است، و حتا برای یافتن کمینه نیز این مطلب درست است، زیرا که ممکن است ما بسیار بدشانس باشیم و همیشه نقطه‌ی تقسیم در کنار بزرگترین عنصر آرایه‌ی فعلی باشد، گرچه به‌خاطر تصادفی بودن این الگوریتم، هیچ ورودی خاصی به‌این بدی نخواهد بود.

ما می‌توانیم یک حد بالا برای $T(n)$، زمان اجرای متوسط Randomized-Select، روی آرایه‌ای از n عنصر را به‌صورت زیر پیدا کنیم. ما در فصل ۸.۴ دیدیم که الگوریتم Randomized-Partition یک تقسیم تولید می‌کند که بخش کوچک‌تر آن به احتمال $2/n$ دارای ۱ عنصر است و به احتمال $1/n$ دارای i عنصر است که $i = 2, 3, \ldots, n-1$. با فرض اینکه یک دنباله‌ی یکنوای صعودی باشد، بدترین حالت Randomized-Select همیشه بدشانس است، یعنی iامین کمینه در بخش بزرگ‌تر نقطه‌ی تقسیم می‌افتد. بنابراین، ما به رابطه‌ی بازگشتی زیر می‌رسیم.

$$
\begin{aligned}
T(n) &\le \frac{1}{n}\left(T(\max(1, n-1)) + \sum_{k=1}^{n-1} T(\max(k, n-k))\right) + O(n) \\
&\le \frac{1}{n}\left(T(n-1) + 2\sum_{k=\lceil n/2 \rceil}^{n-1} T(k)\right) + O(n) \\
&= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n).
\end{aligned}
$$

خط دوم از خط اول به‌واسطه‌ی $\max(1, n-1) = n-1$ نتیجه می‌شود.

$$
\max(k, n-k) = \begin{cases} k & \text{if } k \ge \lceil n/2 \rceil, \\ n-k & \text{if } k < \lceil n/2 \rceil. \end{cases}
$$

اگر n فرد باشد، آنگاه هر کدام از جمله‌های $T(\lceil n/2 \rceil), T(\lceil n/2 \rceil + 1), \ldots, T(n-1)$ دوبار در مجموع ظاهر می‌شوند، و اگر n زوج باشد هر کدام از جمله‌های $T(\lceil n/2 \rceil + 1), T(\lceil n/2 \rceil + 2), \ldots, T(n-1)$ دوبار و جمله‌ی $T(\lceil n/2 \rceil)$ یک بار ظاهر می‌شوند. در هر دو حالت مجموع خط اول از بالا به مجموع خط دوم محدود می‌باشد. خط سوم از خط دوم نتیجه می‌شود البته با استفاده از $T(n-1) = O(n^2)$، و بنابراین جمله‌ی $\frac{1}{n}T(n-1)$ را می‌توان در مقابل $O(n)$ نادیده گرفت.

ما رابطه‌ی بازگشتی را با جای‌گذاری حل می‌کنیم. فرض کنید که برای ثابتی مانند c، داشته باشیم $T(n) \le cn$ که شرایط اولیه‌ی رابطه‌ی بازگشتی را برآورده سازد. با استفاده از فرض استقرا خواهیم داشت

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$
\begin{aligned}
T(n) &\le \frac{2}{n}\sum_{k=\lceil n/2\rceil}^{n-1} ck + O(n) \\
&\le \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2\rceil-1}\right) + O(n) \\
&= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\left\lceil\frac{n}{2}\right\rceil - 1\right)\left\lceil\frac{n}{2}\right\rceil\right) + O(n) \\
&\le c(n-1) - \frac{c}{n}\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right) + O(n) \\
&= c\left(\frac{3}{4}n - \frac{1}{2}\right) + O(n) \\
&\le cn
\end{aligned}
$$

زیرا ما می‌توانیم c به‌قدر کافی بزرگ انتخاب کنیم به‌طوری که $c(n/4 + 1/2)$ از $O(n)$ پیشی بگیرد.

بنابراین هر مرتبه‌ی آماری، و در حالت خاص میانه، را می‌توان با زمان متوسط خطی یافت.

تمرین‌ها

۲-۱

یک نسخه‌ی غیربازگشتی از Randomized-Select را بنویسید.

۲-۲

فرض کنید که ما از Randomized-Select برای یافتن عنصر کمینه در آرایه‌ی $A = \{3, 2, 9, 0, 7, 5, 4, 8, 6, 1\}$ استفاده کنیم. دنباله‌ای از تقسیم‌های متوالی را مشخص کنید که به بدترین حالت کارآیی Randomized-Select منتهی می‌شود.

۲-۳

بیاد بیاورید که در صورت وجود عناصر برابر، رویه‌ی Randomized-Partition زیرآرایه‌ی $A[p, r]$ را به دو زیرآرایه‌ی ناتهی $A[p, q]$ و $A[q + 1, r]$ تقسیم می‌کند به‌طوری‌که هر عنصر از $A[p, q]$ کوچک‌تر یا مساوی هر عنصر از $A[q + 1, r]$ است. اگر عناصر برابر موجود باشند، آیا Randomized-Select درست کار می‌کند؟

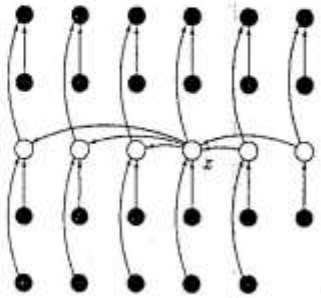
۳.۵.۲ *انتخاب در بدترین حالت زمانی خطی

حال ما یک الگوریتم انتخاب را بررسی می‌کنیم که زمان اجرای آن در بدترین حالت $O(n)$ است. مانند Randomized-Select، الگوریتم Select عنصر دل‌خواه را با تقسیم بازگشتی آرایه ورودی پیدا می‌کند. ولی ایده‌ی اساسی این الگوریتم، تضمین یک تقسیم خوب در هنگام تقسیم شدن آرایه است. Select از الگوریتم تقسیم قطعی Partition ی quicksort (فصل ۸.۱ را ببینید) استفاده می‌کند، البته با کمی تغییر به‌طوری‌که عنصری را که حول آن تقسیم انجام خواهد شد را به‌عنوان پارامتر ورودی بگیرد.

.ذهدی‌م ص‌یخشة ریز ل‌حاره ی‌ارجا اد ار رصنع n زا ی‌دورو ه‌یارآ زا ه‌نیمک رصنع ن‌یما i ،Select م‌یهروگا

۱. n عنصر آرایه‌ی ورودی را به $\lfloor n/5\rfloor$ گروه هر کدام شامل ۵ عنصر و حداکثر یک گروه از $n \bmod 5$ عنصر باقیمانده تقسیم کنید.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۹.۲: انتخاب با زمان $O(n)$.

۲. عنصر میانه‌ی هر $\lceil n/5 \rceil$ گروه را با اجرای الگوریتم insertion sort روی عناصر هر گروه پیدا کنید. (اگر گروه دارای تعداد زوجی عنصر بود، بزرگترین میانه را انتخاب کنید.)

۳. از Select به‌صورت بازگشتی برای یافتن میانه‌ی x از $\lceil n/5 \rceil$ میانه‌ی به‌دست آمده در مرحله دوم استفاده کنید.

۴. با استفاده از یک نسخه‌ی تغییریافته‌ی Partition آرایه‌ی ورودی را حول میانه‌ی میانه‌ها یعنی x تقسیم کنید. فرض کنید k تعداد عناصر در بخش پایین نقطه‌ی تقسیم باشد، پس $n - k$ تعداد عناصر بالای نقطه‌ی تقسیم خواهد بود.

۵. اگر $i \leq k$ از Select به‌صورت بازگشتی برای یافتن i امین عنصر کمینه در بخش پایینی استفاده کنید، و در غیر این‌صورت $(i - k)$ امین عنصر کمینه را در بخش بزرگ‌تر بیابید.

برای بررسی زمان اجرای Select، ما ابتدا یک حد پایین برای تعداد عناصر بزرگ‌تر از عنصر تقسیم، x، پیدا می‌کنیم. شکل ۱ برای تجسم این سامان‌دهی مفید است. حداقل نصف میانه‌هایی که در مرحله دوم پیدا شده‌اند بزرگ‌تر یا مساوی میانه‌ی میانه‌ها یعنی x هستند. بنابراین حداقل $\lceil \lceil n/5 \rceil / 2 \rceil$ گروه ۳ عنصر دارا هستند که بزرگ‌تر از x هستند، مگر گروهی که کمتر از ۵ عنصر را دارد و دلیل این هم بخش‌پذیر نبودن n به ۵ است و هم‌چنین گروهی که x در آن قرار دارد. بدون احتساب این دو گروه، نتیجه می‌گیریم که تعداد عناصر بزرگ‌تر از x حداقل

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

خواهد بود. به‌صورت مشابه، تعداد عناصر کوچک‌تر از x نیز حداقل $3n/10 - 6$ خواهد بود. بنابراین، در بدترین حالت، Select به‌صورت بازگشتی بر روی حداکثر $7n/10 + 6$ عنصر از مرحله‌ی ۵ اعمال خواهد شد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

ما می‌توانیم یک رابطه‌ی بازگشتی برای زمان بدترین حالت اجرای Select، $T(n)$، بدست آوریم. مراحل ۱، ۲، و ۴ زمان $O(n)$ را مصرف می‌کنند. (مرحله‌ی ۲ شامل $O(n)$ بار صدا کردن insertion sort بر روی اندازه‌ی $O(1)$ است.) مرحله‌ی ۳ به اندازه‌ی $T(\lceil n/5 \rceil)$ زمان مصرف می‌کند، و مرحله‌ی ۵ نیز حداکثر $T(7n/10 + 6)$ زمان مصرف می‌کند، البته با فرض اینکه T یکنوا و صعودی است. توجه کنید که برای $n > 140$ داریم $7n/10 + 6 < n$ و همچنین هر ورودی مرکب از ۸۰ یا تعداد کمتری عنصر به زمان $O(1)$ نیاز دارد. بنابراین به رابطه‌ی بازگشتی زیر می‌رسیم

$$T(n) \le \begin{cases} \Theta(1) & \text{if } n \le 80 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n > 80 \end{cases}$$

حال ما با جای‌گذاری نشان می‌دهیم که این زمان اجرا خطی است. فرض کنید که برای یک c مفروض و هر $n \le 80$ داشته باشیم $T(n) \le cn$. با جای‌گذاری این فرض استقرا در سمت راست این رابطه به‌دست خواهیم آورد:

$$\begin{aligned} T(n) &\le c\lceil n/5 \rceil + c(7n/10 + 6) + O(n) \\ &\le cn/5 + c + 7cn/10 + 6c + O(n) \\ &\le 9cn/10 + 7c + O(n) \\ &\le cn, \end{aligned}$$

زیرا ما می‌توانیم برای هر $n > 80$، c را به‌قدر کافی بزرگ اختیار کنیم به‌طوری که $c(n/10 - 7)$ بزرگ‌تر از تابعی باشد که با جمله‌ی $O(n)$ مشخص می‌شود. بنابراین زمان اجرای بدترین حالت Select خطی است.

مانند مرتب‌سازی مقایسه‌ای (بخش ۹.۱ را ببینید)، Select و Randomized-Select اطلاعاتی راجع به مکان نسبی عناصر را فقط با مقایسه پیدا می‌کنند. بنابراین، ویژگی زمان خطی داشتن، نتیجه فرضیاتی راجع به ورودی نیست، درست مانند حالتی که در الگوریتم‌های مرتب‌سازی در فصل ۹ داشتیم. مرتب‌سازی در حالت مقایسه‌ای به زمان $\Omega(n \lg n)$ نیاز دارد، حتی در حالت متوسط (مسئله‌ی ۹-۱ را ببینید)، و بنابراین روش مرتب‌سازی و اندیس‌گذاری که در مقدمه‌ی این بخش آمده بود از نظر مرتبه ناکارآمد است.

تمرین‌ها

۳-۱

در الگوریتم Select، عناصر ورودی را به گروه‌های ۵ تایی تقسیم کردیم. آیا الگوریتم بازهم در زمان خطی اجرا می‌شد اگر آن‌ها را به گروه‌های ۷ تایی تقسیم می‌کردیم؟ به گروه‌هایی ۳ تایی چه‌طور؟

۳-۲

الگوریتم Select را تحلیل کنید و نشان دهید که تعداد عناصر بزرگ‌تر از میانه‌ی میانه‌ها یعنی x، و تعداد عناصر کوچکتر از x حداقل $\lceil n/2 \rceil$ است، البته به‌شرط این‌که $n \ge 38$.

۳-۳

نشان دهید که چه‌گونه می‌توان کاری کرد که الگوریتم quicksort در بدترین حالت با $O(n \lg n)$ اجرا شود.

۳-۴

فرض کنید که یک الگوریتم فقط از مقایسه‌ها برای یافتن i امین عنصر کمینه در یک مجموعه از n عنصر استفاده می‌کند. نشان دهید که این الگوریتم می‌تواند $i - 1$ عنصر کوچکتر و $n - i$ عنصر بزرگتر را نیز بدون انجام مقایسه‌ی اضافی پیدا کند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۵–۳
یک زیربرنامه برای پیدا کردن میانه که در بدترین حالت با زمان خطی عمل می‌کند به صورت یک «جعبه سیاه» داده شده است، یک الگوریتم خطی ساده را برای حل مسئله‌ی انتخاب هر مرتبه‌ی آماری طراحی کنید.

۶–۳
k امین quantileها از یک مجموعه n عضوی، $k - ۱$ مرتبه‌ی آماری هستند که مجموعه‌ی مرتب شده را به k مجموعه با اندازه‌ی برابر تقسیم می‌کنند(با اختلاف یک). یک الگوریتم با زمان $O(n \lg k)$ ارائه دهید که k امین quantileهای یک مجموعه را پیدا کند.

۷–۳
یک الگوریتم با زمان اجرای $O(n)$ پیدا کنید، به‌طوری‌که با گرفتن یک مجموعه S از n عضو متمایز و یک عدد مثبت $k \leq n$، k امین اعداد در S را که به میانه‌ی S نزدیک هستند را بیابد.

۸–۳
فرض کنید که $X[۱..n]$ و $Y[۱..n]$ دو آرایه باشند، که هرکدام حاوی n عدد مرتب شده هستند. یک الگوریتم با زمان اجرای $O(\lg n)$ طراحی کنید که بتواند میانه‌ی تمام $۲n$ عنصر موجود در آرایه‌های X و Y را پیدا کند.

۹–۳
پروفسور Olay در حال همکاری با یک شرکت نفتی است، که این شرکت در حال طراحی یک خط لوله‌ی عظیم است که از شرق به غرب در یک منطقه‌ی نفتی با n چاه می‌گذرد. از هر چاه، یک خط لوله‌ی فرعی که مستقیماً و از نزدیکترین راه (شمال یا جنوب) به لوله‌ی اصلی وصل می‌شود وجود دارد، مانند شکل ۱۰.۲. با در دست داشتن مختصه‌های x و y چاه‌ها، پروفسور محل بهینه‌ی لوله‌ی اصلی را چگونه باید تعیین کند (یعنی باید مجموع طول لوله‌های فرعی کمینه شود؟ نشان دهید که محل بهینه را می‌توان در زمان خطی پیدا کرد.

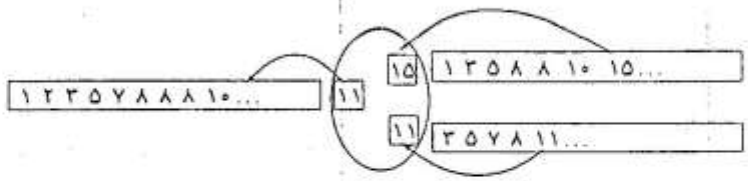
۶.۲ مرتب‌سازی خارجی

در این نوع مرتب‌سازی، تعداد دسترسی به‌دیسک به‌عنوان معیار کارآیی الگوریتم در نظر گرفته می‌شود، چون زمان هر بار دسترسی به دیسک هزاران بار بیشتر از زمان دسترسی به حافظه‌ی اصلی است.

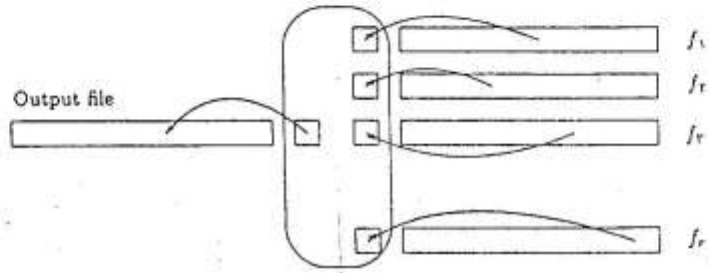
فرض می‌کنیم:

- اطلاعات بر روی فایل‌های به‌صورت ترتیبی ذخیره شده‌است. هر فایل شامل n رکورد است. هر رکورد یک کلید دارد.
- می‌خواهیم فایلی بسازیم که در آن رکوردها براساس کلیدهایشان مرتب باشند.
- با هر دسترسی به دیسک k رکورد خوانده می‌شود.
- تعداد فایل‌هایی که در یک زمان باز هستند r و محدود است.
- تعداد حافظه‌ی اصلی قابل استفاده مستقل از n است.
- عملیات مقایسه و محاسبات دیگر فقط می‌تواند در حافظه‌ی اصلی انجام شود.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۰.۲: ادغام دو فایل مرتب.



شکل ۱۱.۲: ادغام دو فایل مرتب.

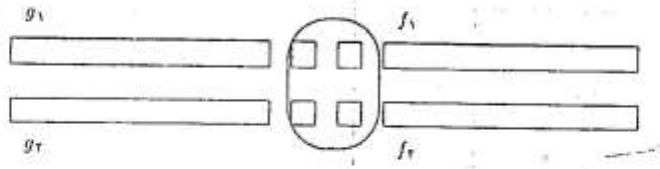
۱.۶.۲ مرتب‌سازی خارجی مبتنی بر ادغام

این روش مبتنی بر ادغام دو قطعه‌ی مرتب از فایل است. فرض کنید دو قطعه‌ی مرتب از فایل بر روی حافظه‌ی خارجی، به‌ترتیب با n و m رکورد داده شده‌اند. می‌خواهیم ادغام‌شده‌ی این دو قطعه را به عنوان قطعه‌ای به طول $n + m$ در حافظه‌ی خارجی بنویسیم. این کار مشابه‌ی عمل ادغام دو آرایه‌ی مرتب انجام می‌شود. در حافظه‌ی اصلی به دو میانگیر با اندازه‌ی یک رکورد نیاز داریم تا بتوان رکوردهای قطعه‌ها را در آن خواند. در ابتدا از هر قطعه یک رکورد می‌خوانیم و هر بار رکوردهای میانگیرها را با هم مقایسه می‌کنیم و رکورد کوچکتر را در انتهای قطعه‌ی خروجی می‌نویسیم و رکورد بعدی قطعه‌ای که رکودش را نوشته‌ایم در همان میانگیر می‌خوانیم. این کار را تا خواندن و نوشتن همه‌ی رکوردهای دو قطعه‌ی ورودی ادامه می‌دهیم.

ادغام دو قطعه‌ی مرتب

اگر قطعه‌ی اول n_1 رکورد و قطعه‌ی دوم n_2 رکورد داشته باشند و $k = 1$، با $n_1 + n_2$ بار خواندن و همین تعداد نوشتن می‌توان ادغام را انجام داد. این کار در حالت کلی با $\lceil \frac{n_1}{k} \rceil + \lceil \frac{n_2}{k} \rceil$ بار دسترسی به دیسک قابل انجام است. مقدار حافظه‌ی مورد نیاز در این حالت به‌اندازه‌ی $3k$ رکورد می‌باشد. (به شکل ۱۰.۲ توجه کنید.)،

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۲.۲: الگوریتم MergeSort خارجی

ادغام r قطعه‌ی مرتب

با همین روش می‌توانیم r قطعه‌ی مرتب از فایل‌های مختلف را با هم ترکیب کنیم. (به شکل ۱۱.۲ توجه کنید.) تعداد دست‌رسی‌هابرابر $2\sum_{i=1}^{r}\lceil \frac{n_i}{k} \rceil$ بار و مقدار حافظه‌ی مورد نیاز: به‌اندازه‌ی $k(r+1)$ رکورد.

الگوریتم کلی

برای $k = 1$ بیان می‌شود. چهار فایل f_1، f_2، g_1 و g_2 احتیاج است.

۱. فایل ورودی را به دو فایل f_1 و f_2 با حداکثر تعداد یک رکورد اختلاف تقسیم کن.

۲. برای $i = 1, \ldots, M$ مراحل زیر را تکرار کن:

در این مرحله فرض می‌کنیم که f_1 و f_2 (یا g_1 و g_2) شامل قطعاتی به طول 2^{i-1} هستند و هر قطعه مرتب است و تعداد قطعات دو فایل ورودی حداکثر یک واحد اختلاف دارد.

۱–۲) f_1 و f_2 را به‌صورت فایل‌های ورودی در نظر می‌گیریم. قطعات با شماره‌های یکسان f_1 و f_2 را با یکدیگر ادغام کن و قطعه‌ای به طول دو برابر ایجاد کن. حاصل این ادغام قطعاتی مرتب به‌طول 2^i (بجز حداکثر یک قطعه به‌طول کمتر) است این قطعات را به‌ترتیب یک‌بار در g_1 و بار دیگر در g_2 بنویس.

۱–۳) g_1 و g_2 را به عنوان فایل‌های ورودی و f_1 و f_2 را به عنوان فایل‌های خروجی در نظر بگیر و مرحله‌ی بالا را تکرار کن.

- با استقرا می‌توان نشان داد که در انتهای مرحله‌ی i هر فایل خروجی دارای قطعه‌هایی مرتب و به طول 2^i است، بجز حداکثر یک قطعه که طولش از 2^i کمتر است. همچنین تعداد قطعه‌های دو فایل خروجی حداکثر یک واحد اختلاف دارند.
- بنابراین برای $M = \lceil \log n \rceil$ (تعداد تکرار حلقه) یکی از فایل‌های خروجی حاوی یک قطعه‌ی مرتب شامل تمام n رکورد فایل ورودی و دیگری خالی است.
- با توجه به‌این‌که در هر تکرار همه‌ی n رکورد یک‌بار خوانده و یک‌بار نوشته می‌شوند، تعداد دسترسی به دیسک در مجموع برابر $2n(\lceil \log n \rceil + 1)$ است ($2n$ بار خواندن و نوشتن برای تقسیم فایل اصلی).

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

- برای حالت $k > 1$، این تعداد برابر $2\lceil \frac{n}{k} \rceil (\lceil \log n \rceil + 1)$ خواهد بود.
- با تقسیم فایل ورودی به r فایل با اندازه‌هایی یکسان و با استفاده از r حافظه نیز می‌توان فایل را مرتب کرد در آن صورت، تعداد دسترسی به دیسک $2n\lceil \log_r n \rceil + 2n$ می‌شود و در حالت کلی برابر $2\lceil \frac{n}{k} \rceil (\lceil \log_r n \rceil + 1)$.
- در حالت کلی به حافظه‌ای به‌اندازه‌ی $(r+1)k$ نیاز است.
- حالت کلی را Multiway Merge می‌گویند.

مثال:

```
f1  28 3 93 10 54 65 30  90 10  69  8  22
f2  31 5 96 40 85  9 39  13  8  77 10

g1  28 31 | 93 96 | 54 85 | 30 39 |  8 10 |  8 10
g2   3  5 | 10 40 |  9 65 | 13 90 | 69 77 | 22

f1  3  5 28 31 | 9 54 65 85 | 8 10 69 77
f2 10 40 93 06 |13 30 39 90 | 8 10 22

g1  3  5 10 28 31 40 93 96 | 8 8 10 10 22 69 77
g2  9 13 30 39 54 65 85 90 |

f1  3 5  9 10 13 28 39 31 39 40 54 65 85 90 93 96
f2  8 8 10 10 22 69 77

g1  3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 ..
```

۲.۶.۲ مرتب‌سازی خارجی Polyphase

- سه (در حالت کلی به $r+1$) فایل لازم دارد.
- به فایل ورودی به کم‌ترین تعداد رکورد با بیشترین کلید اضافه کن تا n برابر F_i (iامین عدد فیبانوچی) شود.
- فایل ورودی را به دو فایل با اندازه‌های F_{i-1} و F_{i-2} تقسیم کن $(F_i = F_{i-1} + F_{i-2})$
- باندازه‌ی $M = ?$ بار تکرار کن
  - فرض کنید از سه‌فایل f_1 دارای F_m قطعه‌ی مرتب (در ابتدا $F_m = F_i$) است که اندازه‌ی هر قطعه F_r است (در ابتدا $F_r = F_0 = 1$).
  هم‌چنین f_2 دارای F_{m-1} قطعه‌ی مرتب است که اندازه‌ی هر قطعه F_{r+1} است. و f_3 خالی است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

— F_m قطعه‌ی مرتب از فایل f_1 را با همین تعداد قطعه‌ی مرتب از فایل f_2 را با هم ادغام کن و حاصل را در f_3 بنویس.

— حالا f_1 خالی، f_3 دارای $F_{m-1} = F_{m+1} - F_m$ قطعه به اندازه‌ی F_{r+1} و f_2 دارای F_m قطعه‌ی مرتب به اندازه‌ی $F_{r+2} = F_r + F_{r+1}$ است.

— نام‌گذاری فایل‌ها را به تناسب تغییر بده.

مثال: $n = 34$

after pass	f1	f2	f3
initial	13(1)	21(1)	-
1	-	8(1)	13(2)
2	8(3)	-	5(2)
3	3(3)	5(5)	-
4	-	2(5)	3(8)
5	2(13)	-	1(8)
6	1(13)	1(21)	-
7	-	-	1(34)

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

فصل ۳

داده‌ساختارها

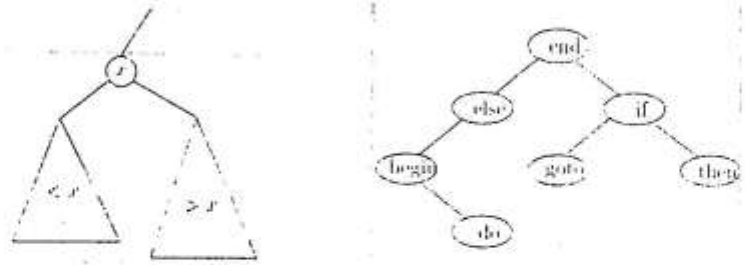
۱.۳ دسته‌بندی

۱) داده‌ساختارهای ساده

- لیست‌ها (لیست‌های ساده‌ی یک‌طرفه، دوطرفه، دوار)
- پشته‌ها
- صف‌ها
- درخت‌ها

۲) داده‌گونه‌ی انتزاعی مجموعه‌ها

- فرهنگ داده‌ای (درج، حذف و جست‌وجو)
  - جدول پراکندگی (hash table) : باز، بسته، متوسط $O(1)$
  - درخت دودویی جست‌وجو (BST): حداکثر $O(n)$ و متوسط $O(\lg n)$
  - درخت دودویی جست‌وجوی با ارتفاع $O(\lg n)$
    - a) درخت دودویی جست‌وجوی متوازن
    - b) درخت AVL (درختی که اختلاف ارتفاع دو زیردرخت هر گره‌ی آن حداکثر ۱ باشد)
    - c) درخت «قرمز–سیاه» Red-Black tree
  - درخت‌های کاملاً متوازن (ارتفاع $O(\lg n)$)
    - a) درخت ۳–۲
    - b) درخت «بی»
- صف اولویت (درج و حذف کوچک‌ترین عنصر): $O(\lg n)$
- Disjoint Find-Merge (پیداکردن، ادغام)



شکل ۱.۳: درخت دودویی جست‌وجو و یک مثال.

۲.۳ داده‌ساختارها برای فرهنگ داده‌ای

۱.۲.۳ درخت دودویی جست‌وجو

تعریف: درختی دودویی که کلیه‌ی عناصر زیر درخت چپ گره‌ی r کمتر از r و کلیه‌ی عناصر زیر درخت راست آن بیش‌تر از r باشد. (شکل ۱.۳)

پیمایش بین‌ترتیب یک درخت دودویی جست‌وجو مرتب‌شده‌ی عناصر را تولید می‌کند.

ترتیب درج در ارتفاع درخت حاصل بسیار مؤثر است (شکل‌های ۲.۳ و ۳.۳)

$$\lceil \lg n \rceil \le \text{ارتفاع} \le n - 1 \quad \text{(چرا؟)}$$

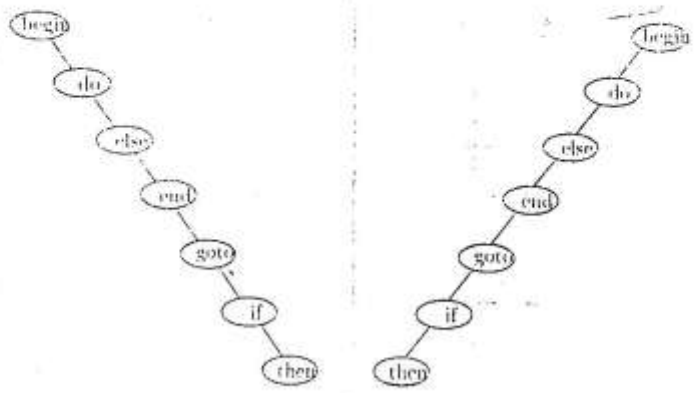
متوسط ارتفاع: $O(\lg n)$

تعداد درخت‌های دودویی جست‌وجو که با $n_1 < n_2 < \cdots < n$ می‌توان ساخت:

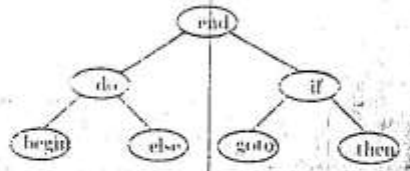
$$\begin{aligned} T(n) &= \sum_{i=1}^{n} T(i-1)T(n-i), \quad T(0) = 1 \\ &= \frac{1}{n+1}\binom{2n}{n} \end{aligned}$$

عدد کاتالان

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲.۳: درخت دودویی جست‌وجو با بیش‌ترین ارتفاع.



شکل ۲.۳: درخت دودویی جست‌وجو با کم‌ترین ارتفاع.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

جست‌وجو

```
FUNCTION MEMBER (x: LabelType; A: BST): BOOLEAN;
begin
   if A = nil then return (false);
   if A^.label = x then return (true);
   if x < A^.label then
        return (member(x, A^.left))
   else return (member(x, A^.right))
```

درج

```
PROCEDURE INSERT (x: LabelType; var A: BST);
begin
   if A = nil then begin
        new(A); A^.label := x;
        A^.left := nil;
        A^.right := nil;
   end else  if x < A^.label then
                insert (x,A^.left)
             else  if x > A^.label then
                      insert (x, A^.right)
end;
```

حذف کوچک‌ترین عنصر

```
FUNCTION DELETEMIN (var A:BST ): LabelType;
var t: BST;
begin
   if A = nil then error ('no such elment');
   if A^.left = nil then begin
        deletemin := A^.label;
        t := A; A := A^.right;
        dispose(t);
   end
   else return (deletemin(A^.left))
end;
```

حذف

```
PROCEDURE DELETE (x:labeltype; var A:BST);
var t: BST;
begin
   if a = nil then error ('tree is empty');
   if x < A^.label then delete(x,A^.left)
   else if x > A^.label then delete(x,A^.right)
        else  begin (A^.label=x)
             t:=A;
             if A^.left = nil then begin
                  A:=A^.right;
                  dispose (t);
             end
             else if A^.right = nil then begin
                       A:=A^.left;
                       dispose (t);
                  else A^.label:=deletemin(A^.right)
        end
end;
```

۲.۲.۳ متوسط ارتفاع درخت دودویی جست‌وجو

اگر $n_1 < n_2 < \cdots < n_n$ به صورت تصادفی و با احتمال یکسان وارد یک درخت دودویی جست‌وجوی تهی T شوند. اثبات می‌کنیم که متوسط ارتفاع T برابر $O(\log n)$ است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

- فرض کنید n_1 اولین عنصری است که درج می‌شود.
- n_1 ریشه‌ی درخت خواهد بود.
- n_2 تا n_{i-1} به هر ترتیبی که درج شوند در زیردرخت چپ قرار خواهند گرفت.
- n_{i+1} تا n_n در زیر درخت راست خواهند بود.
- اگر $h(n)$ متوسط ارتفاع T باشد، داریم:

$$h(n) = 1 + \sum_{i=1}^{n} \frac{1}{n} \max\{h(i-1), h(n-i)\}$$

- فرض می‌کنیم $h(n) \le c \log n$ برای یک $c > 0$.
- با توجه به این‌که $h(i)$ یک تابع غیرنزولی است،

$$\begin{aligned}
h(n) &= 1 + \frac{1}{n}\left(\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} h(n-i) + \sum_{i=\lfloor \frac{n}{2} \rfloor - 1}^{n} h(i-1)\right) \\
&\le 1 + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor - 1}^{n} h(i-1) \\
&\le 1 + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} h(i) \\
&\le 1 + \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} c \log i \\
&\le 1 + \frac{2c}{n} \log \frac{(n-1)!}{(\frac{n}{2}-1)!} \\
&\le 1 + \frac{2c}{n} c[\log n! - \log(\frac{n}{2})!]
\end{aligned}$$

- با توجه به این که $\log n! = \Theta(n \log n)$ یعنی $k_1 n \log n \le \log n! \le k_2 n \log n$،

$$\begin{aligned}
h(n) &\le 1 + \frac{2c}{n}(k_2 n \log n - k_1 \frac{n}{2} \log \frac{n}{2}) \\
&\le 1 + 2c(k_2 - \frac{k_1}{2}) \log n - 2ck_1
\end{aligned}$$

می‌توان $0 < k_1 < k_2$ را پیدا کرد که $k_2 - \frac{k_1}{2} < 1/2$ و $1 - 2ck_1 \le 0$ پس

$$h(n) \le c \log n$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۳.۳ صف اولویّت (Priority Queue)

صف اولویّت یک داده‌ساختار است برای نگه‌داری عناصر یک مجموعه بکار می‌رود به‌طوری‌که هر یک از عناصر یک کلید دارد که نمایانگر اولویّت آن عنصر می‌باشد. اعمال مهمی که بر روی یک صف اولویت انجام می‌شوند عبارتند از:

- Insert(Q,x): عنصر x را در صف Q وارد می‌کند.
- Maximum(Q): عنصر با اولویّت بیشتر را برمی‌گرداند.
- Exteract-Max(Q): عنصر با اولویّت بیشتر را از صف حذف و برمی‌گرداند.

با توجه به شباهت میان heap و صف اولویّت می‌توان اعمال فوق را با ساختار heap پیاده‌سازی کرد. در این رابطه رویه‌های زیر پیاده‌سازی می‌شوند:

- Heap-Insert (A, key): دراین رویه یک عنصر به Heap اضافه می‌شود. برای انجام این کار ابتدا یک برگ جدید به درخت (در جای لازم) با کلید key اضافه می‌شود و سپس key تا قرار گرفتن در مکان مناسب به سمت بالا حرکت می‌کند.

```
Heap_Insert(A,key)
        Heap_size(A) <- Heap_size(A)+1
        while(i>1 and A[parent(i)]<key)
             do    A[i] <- A[parent(i)]
                   i <- parent(i)
        A[i] <- key
```

زمان اجرا: از آن‌جایی که طول مسیر پیمایش شده حداکثر ارتفاع heap است. $T(n) = O(\log n)$.

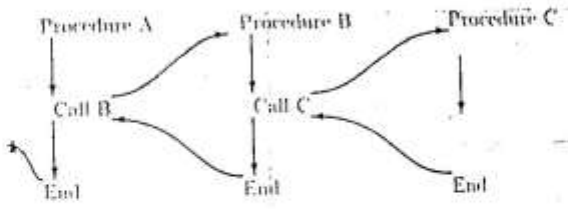
- Heap-Maximum (A): عنصر با اولویّت بیش‌تر را که در A[1] قرار دارد برمی‌گرداند.
- Heap-Extract-Maximum (A): عنصر با اولویّت بیش‌تر را از heap حذف کرده و مقدار آن را برمی‌گرداند:

```
Heap_Extract_Max(A)
        if heap_size(A)<1
           Then  error "Heap underflow"
        max <- A[1]
        A[1] <- A[heap_size(A)]
        heap_size(A) <- heap_size(A)-1
        Heapify(A,1)
        return max
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

واضح است که زمان اجرای این رویه $T(n) = O(\log n)$ است. در نتیجه می‌توان یک صف اولویّت را با استفاده از heap پیاده‌سازی کرد و هزینه‌ی اعمال فوق حداکثر $O(\log n)$ خواهد بود.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴.۳: انتقال کنترل برنامه در فراخوانی و بازگشت

۴.۳ تبدیل الگوریتم‌های بازگشتی به غیربازگشتی

الگوریتم‌های بازگشتی شامل دو مرحله‌ی مهم هستند:

۱. عمل فراخوانی

۲. بازگشت از یک فراخوانی

هدف آن است که بتوانیم دو مرحله‌ی فوق را شبیه سازی نماییم. فراخوانی مانند وقفه عمل می‌کند. کلیه‌ی متغیرها در لحظه فراخوانی ذخیره می‌شوند، عملیاتی صورت می‌گیرد و هنگام بازگشت وضعیت پیش از فراخوانی مجدداً بازسازی می‌شود.

در نقطه H مقادیر متغیرها(به جز متغیرهای از نوع آدرسی Called By Reference) همان مقدار قبل از فراخوانی را دارند. فراخوانی می‌تواند به رویه‌های دیگر و یا به خود همان رویه (در برنامه‌های بازگشتی) باشد . هر فراخوانی ممکن است نتیجه‌ای به همراه داشته باشد ولی مقدار متغیرها تغییر نمی‌کند.

هر فراخوانی (Call) شامل مراحل زیر است :

۱. کلیه‌ی متغیرهای محلی (در حالت کلی کلیه متغیرهای دسترس‌پذیر) و مقدارهایشان در پشته‌ی سیستم قرار می گیرند (Push).

۲. آدرس بازگشت به پشته منتقل می شود (Push).

۳. عمل انتقال پارامترها (Parameter Passing) صورت می‌گیرد. پارامترها ممکن است از نوع ارزشی (Val) یا آدرسی (Variable) باشند.

۴. کنترل برنامه (ثبات شمارنده‌ی برنامه Program Counter) به ابتدای رویه‌ی جدید اشاره می‌کند.

عمل بازگشت (Return) عکس عملیات فوق را انجام میدهد.

۱. مقدارهای متغیرهای محلی را از رکورد بالای پشته برداشته و در خودشان قرار می‌دهیم.

۲. آدرس بازگشت را از بالای پشته به دست می آوریم.

۳. آخرین رکورد را از پشته برمی‌داریم (Pop).

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۴. کنترل برنامه را از آدرس بازگشت (بند ۲) ادامه می‌دهیم.

برای روشن شدن این مطلب به مثال زیر توجه می‌کنیم. در مساله‌ی برج‌های هانوی، هدف انتقال n سکه‌ی سوراخ‌دار از میله‌ی (یا برج) مبدا به میله‌ی مقصد با کمک میله‌ی سوم است. در انتقال سکه‌ها دو نکته باید رعایت شود:

۱. هر بار بالاترین سکه باید حرکت داده شود.

۲. سکه‌ی بزرگتر بر روی سکه‌ی کوچکتر قرار نگیرد.

خروجی برنامه گام‌های انتقال سکه‌ها از میله‌ی مبدا به میله‌ی مقصد است. برای حل مساله میله‌ی مبدا را f (from) و میله‌ی مقصد را t (to) و میله‌ی کمکی را h (help) می‌نامیم. برنامه‌ی انتقال سکه‌ها به طور بازگشتی به صورت زیر است:

```
procedure Hanoi(n,f,t,h:Integer);
begin
   If (n = 1) Then
        writeln(f,'->',t)
   else begin
        Hanoi(n-1,f,h,t);
A ->
        writeln(f,'->',t);
B ->
        hanoi(n-1,h,t,f)
   end
end;
```

مرحله‌ی ۱: دراین مرحله یک فراخوانی انجام می‌شود. ارزش متغیرها (h=2, t=3, f=1, n=3) وآدرس محل فراخوانی (Ret_Addr='A') دربشته ذخیره شده و متغیرها ارزش جدیدی پیدا می‌کنند. (h=3, t=2, f=1, n=2). و یک پرش به ابتدای برنامه انجام می‌شود. (شکل ۵.۳ (a))

مرحله‌ی ۲: یک فراخوانی مستقل دیگر صورت می‌گیرد. و ارزش متغیرها و آدرس محل فراخوانی (A)دربشته قرار میگیرد. مقادیر متغیرها عبارت است از:

n = 1, f = 1, t = 3, h = 2

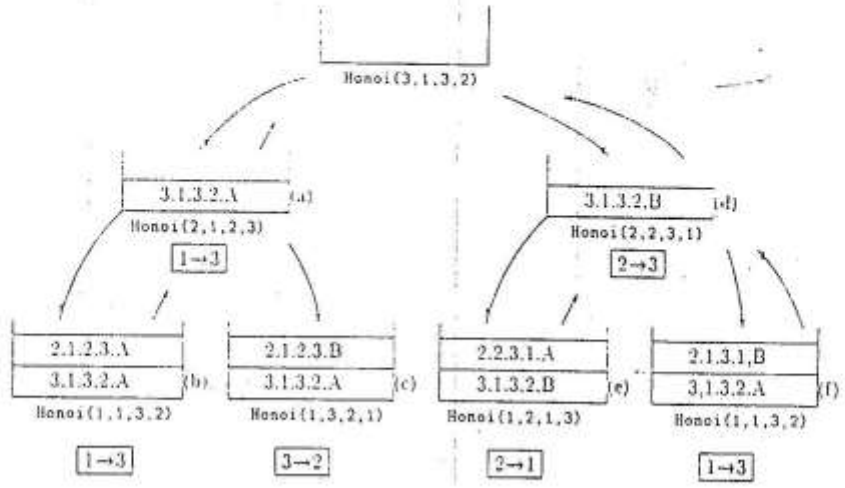
و یک پرش به ابتدای برنامه انجام می‌شود.(شکل ۵.۳ (b))

مرحله‌ی ۳: چون n = 1 است قسمت اول برنامه اجرا می‌شود. (خروجی 1 -> 3)

مرحله‌ی ۴: یک بازگشت (Return) صورت می‌گیرد.مقادیر متغیرهای محلی وآدرس بازگشت از پشته برداشته می‌شوند یعنی داریم:

n = 2 , f = 1 , t = 2 , h = 3, Ret_Addr = 'A'

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۵.۳: مراحل اجرای فراخوانی Hanoi(3,1,3,2).

آخرین رکورد پشته را (Pop) می‌کنیم و کنترل برنامه‌را از آدرس بازگشت ('A') با مقدارهای جدید متغیرها ادامه می‌دهیم .(شکل ۵.۳ (a). خروجی 2 -> 1)

مرحله‌ی ۵: یک فراخوانی مستقل دیگرانجام می‌شود. مقدارهای متغیرها وآدرس بازگشت (Ret_Addr='B') درپشته ذخیره شده و پس ازانجام عمل انتقال پارامترها داریم (شکل ۵.۳ (b)):

```
n = 1, f = 3 , h = 2 , t = 1
```

مرحله‌ی ۶: n = 1 پس فراخوانی صورت نمی‌گیرد. (خروجی 2 -> 3)

مرحله‌ی ۷: عمل بازگشت پس از طی قدم‌های زیر صورت می‌گیرد:

۱. ارزش متغیرها را از بالای پشته برداشته و داریم:

```
n = 2, f = 1, t = 2, h = 3
```

۲. آدرس محل فراخوانی ('B') را از پشته بر می‌داریم.

۳. پشته را (Pop) می‌کنیم.

۴. برنامه را از ('B') با ارزش‌های جدید متغیرها دنبال می‌کنیم. (شکل ۵.۳ (c))

مرحله‌ی ۸: یک بازگشت دیگر صورت می‌گیرد و داریم:

```
n = 3, f = 1, t = 3, h = 2, Ret_Addr = 'A'
```

برنامه را از ('A') پی می‌گیریم .(خروجی 3 -> 1)

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مرحله‌ی ۹: پس از انجام عمل فراخوانی در پشته یک رکورد جدید با مقادیر h = 2, Ret_Addr = 'A' n = 3, f = 1, t = 3, داریم و ارزش جدید متغیرها عبارتند از (شکل ۵.۳ (d)):

n = 2 , f = 2 , t = 3 , h = 1

مرحله‌ی ۱۰: یک فراخوانی انجام می‌گیرد و ارزش جدید متغیرها عبارتند از: n = 1, f = 2, t= 1, h = 3 (شکل ۵.۳ (e))

مرحله‌ی ۱۱: چون n = 1، فراخوانی دیگری در این مرحله روی نمی‌دهد. (خروجی 1 -> 2)

مرحله‌ی ۱۲: یک بازگشت صورت می‌گیرد، و مقدارهای جدید متغیرها به صورت زیر است:

n = 3, f = 2, t = 3, h = 1, Ret_Addr = 'A'

برنامه را از('A') دنبال می‌کنیم . (شکل ۵.۳ (d) ، خروجی 3 -> 2)

مرحله‌ی ۱۳ فراخوانی مستقل داریم. مقدارهای متغیرها و آدرس محل فراخوانی (Ret_Addr='B') را در پشته وارد می‌کنیم و ارزش جدیدی به متغیرها می‌دهیم (n=1, f=1, t=3, h=2). (شکل ۵.۳ (f))

مرحله‌ی ۱۴ n = 1 فراخوانی دیگری صورت نمی‌گیرد. (خروجی 3 -> 1)

مرحله‌ی ۱۵ در این مرحله یک بازگشت رخ می‌دهد (شکل ۵.۳ (d)).

n = 2, f = 2, t = 3, h = 1

مرحله‌ی ۱۶ یک بازگشت داریم.

n = 3, f = 1, t = 3, h = 2

برنامه را از('B') دنبال می‌کنیم .

مرحله‌ی ۱۷ پایان اجرای رویه و بازگشت به برنامه‌ی اصلی.

۵.۳ برنامه‌ی غیر بازگشتی

رویه‌ی برج‌های هانوی به صورت غیر بازگشتی (Non-Recursive) به صورت زیر است:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Const  Max = 100;
Type   Stack = ..

Procedure NR_Hanoi(n,f,t,h:Integer);
Var
   s : Stack;
   Rot_Addr : char;
   Label 1,99;
Begin
    MakeNull(s);
1:  If (n = 1) Then Begin
        WriteLn(f,'->',t);
        Goto 99
    End
    Else Begin
        Push(s,n,f,t,h,'A')
        n := n-1
        Swap(h,t);
        Goto 1
    End;
99: If Empty(s) Then  Exit;
    Ret_Addr, h, t, f, n  <-  Top(s);
    Pop(s);
    Case Ret_Addr Of
      'A': Begin
               WriteLn(f,'->',t);
               Push(s,n,f,t,h,'B');
               n := n-1;
               Swap(h,f);
               Goto 1;
             End;
      'B': Goto 99
     End
End;
```

در این برنامه یک متغیر با گونه‌ی (Stack) برای شبیه‌سازی پشته تعریف شده است. آدرس بازگشت فراخوانی از گونه‌ی نوبسه است. Label برای مشخص نمودن محل پرش‌ها ست.

ابتدا برنامه‌ی پشته (s) را برای تشخیص ختم بازگشت تهی می‌کنیم (MakeNull). اگر (N=1) یک پرش به‌انتهای برنامه صورت می‌گیرد (شبیه سازی عمل Return) و درغیر این صورت اولین فراخوانی باید شبیه‌سازی شود: به این ترتیب که ابتدا کلیه‌ی متغیرهای محلی (s,n,f,t,h) و آدرس بازگشت ('A') در پشته قرار می‌گیرد. سپس عمل انتقال پارامترها انجام می‌شود وسرانجام یک پرش به اول برنامه صورت می‌گیرد. اگر پشتهٔ خالی باشد، یک بازگشت به برنامه‌ی اصلی صورت می‌گیرد و در غیر این صورت متغیرهای محلی و آدرس بازگشت مقدار قبلی خود را پیدا می‌کنند و بالاترین رکورد پشته برداشته (Pop) می‌شود. بسته به مقدار آدرس بازگشت دو کار زیر صورت می‌گیرد:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱. فراخوانی دوم شبیه سازی می‌شود.

۲. پرش انجام می‌شود.

برنامه‌ی بالا نمونه‌ای از یک تبدیل برنامه بازگشتی به غیربازگشتی است، در این برنامه اصول برنامه نویسی چندان رعایت نشده ولی می‌توان آن را بهینه نمود.

۱.۵.۳ حذف آخرین بازگشت (Tail Recursion)

آخرین فراخوانی بازگشتی که بعد از آن در هیچ شرایطی دستوری که از مقدارهای متغیرها استفاده کند، اجرا نشود را آخرین بازگشت میگوییم. این بازگشت را می‌توان بدون استفاده از پشته حذف کرد. برنامه‌ی زیر نمونه‌ای از آخرین بازگشت است.

```
Procedure A
Begin
..
..
   Call A
x ->
End;
```

در بازگشت به این فراخوانی (A) متغیرهای محلی مقدارهایشان تغییر می‌کند واجرای برنامه از نقطه‌ی (x) دنبال می‌شود. ولی (x) تنها یک بازگشت است. بنابراین می‌توان عمل وارد کردن ارزش متغیرها و آدرس فراخوانی در پشته (Push) و حذف از پشته (Pop) را از برنامه حذف نمود و فراخوانی تنها به یک انتقال پارامتر و یک پرش به اول برنامه تبدیل شود. به عنوان مثال برنامه‌ی برج‌های هانوی را می‌توان به صورت زیر نوشت:

```
Procedure Hanoi(n,f,t,h:Integer);
Label 1;
Begin
 1: If (n = 1) Then  WriteLn(f,'->',t)
    Else Begin
         Hanoi(n-1,f,h,t);
         WriteLn(f,'->',t);
         n := n-1            {parameter passing}
         Swap(f,h);          {parameter passing}
         GoTo 1
    End
End;
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

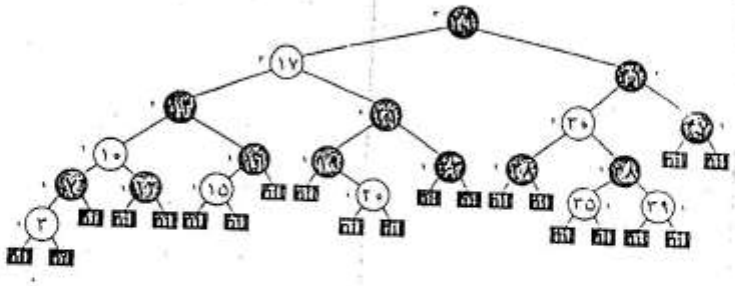
اگر برنامه‌ی فوق به غیر بازگشتی تبدیل شود، چون فقط یک آدرس بازگشت داریم، دیگر نیازی به نگه‌داری آدرس بازگشت نیست. رویه‌ی فوق به صورت زیر غیر بازگشتی می‌شود:

```
cedure NR_Hanoi2(n,f,t,h:Integer);
Var s : Stack;
Label 1,99;

Begin
    MakeNull(s);
1:  If (n = 1) Then Begin
          WriteLn(f,'->',t);
          Goto 99
    End Else
    Begin
       Push(s,n,f,t,h)
       n := n-1
       Swap(h,t);
       Goto 1
    End;
99: If Empty(s) Then Exit;
     n,f,t,h <- Top(s);
     Pop(s);
     WriteLn(f,'->',t);
     n := n-1;
     Swap(h,f);
     Goto 1;
End;
```

دربرنامه‌ی برج‌های هانوی همه‌ی متغیرها از نوع ارزشی (Value) بودند. اما اگرمتغیرها از نوع (Variable) باشند چه باید کرد؟ در این صورت می‌توانیم آدرس این متغیرها را به جای مقادیرشان در پشته وارد کنیم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۶.۳: مثال درخت قرمز– سیاه

۶.۳ درخت قرمز– سیاه (Red-Black Tree)

هدف: ایجاد درخت جستجوی دودویی که موارد زیر را تضمین نماید:

۱. $h = O(\lg n)$ که n تعداد گره‌ها و h ارتفاع درخت می‌باشد. واضح است که در اینصورت عمل جستجوی یک عنصر در فرهنگ داده‌ای «در بدترین حالت» $O(\lg n)$ خواهد بود.

۲. بتوانیم عملیات درج و حذف در فرهنگ داده‌ای را با الگوریتمی با پیچیدگی $O(\lg n)$ انجام دهیم بگونه‌ای که خاصیت ۱ حفظ شود.

برای رسیدن به اهداف فوق درخت قرمز– سیاه را تعریف می‌کنیم:

تعریف: درخت قرمز– سیاه، درخت جستجویی می‌باشد که هر عنصر آن یک مؤلفه‌ی رنگ (color) دارد که می‌تواند سیاه یا قرمز باشد. اشاره‌گرهای nil در این درخت را برگهای nil و سایر عناصر شامل برگهای درخت اولیه را گره‌های داخلی (Internal Nodes) می‌نامیم. نمونه‌ای از درخت قرمز– سیاه در «شکل ۶.۳» آورده شده است.

۱.۶.۳ خواص درخت قرمز– سیاه:

۱. هر گره یا قرمز است و یا سیاه.

۲. هر برگ nil سیاه است.

۳. دو فرزند یک گره قرمز، سیاهند (پدر یک گره قرمز نمی‌تواند قرمز باشد).

۴. هر مسیر ساده از یک گره به برگ فرزند (نه لزوماً فرزند مستقیم) شامل تعداد یکسانی گره سیاه می‌باشد.

۵. ریشه درخت سیاه است (این شرط، از شروط اساسی نیست.)

تعاریف و قضایای اولیه:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱. سیاه‌ارتفاع گرهٔ x: بنابه تعریف $bh(x)$[1] برابر است با تعداد گره‌های سیاه از x تا یک برگ فرزند[2].

۲. عموی گره x:

```
if parent[x] = right[parent[parent[x]]]
   then uncle[x] := left[parent[parent[x]]]
else uncle[x] := right[parent[parent[x]]]
```

قضیه ۱ حداکثر ارتفاع یک درخت قرمز– سیاه که دارای n گره داخلی می‌باشد، برابر $2\lg(n+1)$ است.

برای اثبات قضیه فوق ابتدا نشان می‌دهیم که یک زیردرخت به ریشه دلخواه x حداقل دارای $2^{bh(x)}-1$ گره داخلی می‌باشد. برای اثبات از استقراء بر روی ارتفاع گره داخلی x در درخت استفاده می‌نماییم.

پایه: اگر ارتفاع x صفر باشد، x حتماً برگ و در نتیجه nil خواهد بود. بنابراین زیردرخت به ریشه x حداقل $2^{bh(x)}-1$ یعنی $2^0-1=0$ گره داخلی دارد (بدیهی).

گام استقرائی: گره داخلی x را در نظر بگیرید. سیاه– ارتفاع x عددی مثبت می‌باشد و x دارای دو فرزند می‌باشد. هر کدام از فرزندان برحسب اینکه قرمز باشند یا سیاه، سیاه– ارتفاعی برابر $bh(x)$ یا $bh(x)-1$ خواهند داشت. به علت اینکه هر فرزند x ارتفاعی کمتر از خود x دارد می‌توانیم با استفاده از فرض استقراء نتیجه بگیریم که هر زیر درخت به ریشهٔ یک فرزند x حداقل $2^{bh(x)-1}-1$ گره داخلی دارد. بنابراین زیردرخت به ریشهٔ x حداقل $2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$ گره داخلی خواهد داشت.

از طرف دیگر می‌دانیم که در درخت به ارتفاع h حداقل نیمی از گره‌ها (بدون درنظر گرفتن ریشه) بر روی هر مسیر ساده از ریشه به برگ، سیاه هستند. زیرا در غیراینصورت خاصیت ۳ از خواص درخت قرمز– سیاه نقض خواهد شد. نتیجتاً سیاه– ارتفاع ریشهٔ درخت حداقل $\frac{h}{2}$ خواهد بود. بنابراین:

$$n \geq 2^{\frac{h}{2}}-1 \Rightarrow 2^{\frac{h}{2}} \leq n+1 \Rightarrow h \leq 2\lg(n+1)$$

نتیجه: ارتفاع درخت قرمز– سیاه $O(\lg n)$ است.

۲.۱.۳ دوران

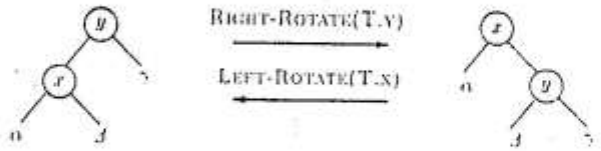
برای بازیابی خواص درخت قرمز– سیاه بعد از عملیات درج و حذف در بعضی شرایط مجبور خواهیم شد که رنگ بعضی از گره‌ها را عوض کنیم یا تغییراتی در ساختار اشاره‌گرها اعمال نمائیم. به همین منظور دو عمل دوران (راستگرد و چپگرد) تعریف می‌نمائیم. با توجه به «شکل ۷.۳»، واضح است که این دو عمل هیچ اشکالی در خواص یک درخت دودوئی ایجاد نخواهد نمود. شکل ۸.۳ مثالی از LEFT-ROTATE (T,x) را نشان می‌دهد.

نکته: وقتی ما یک دوران چپگرد انجام می‌دهیم فرض می‌کنیم که فرزند راست گرهٔ مورد نظر nil نباشد.

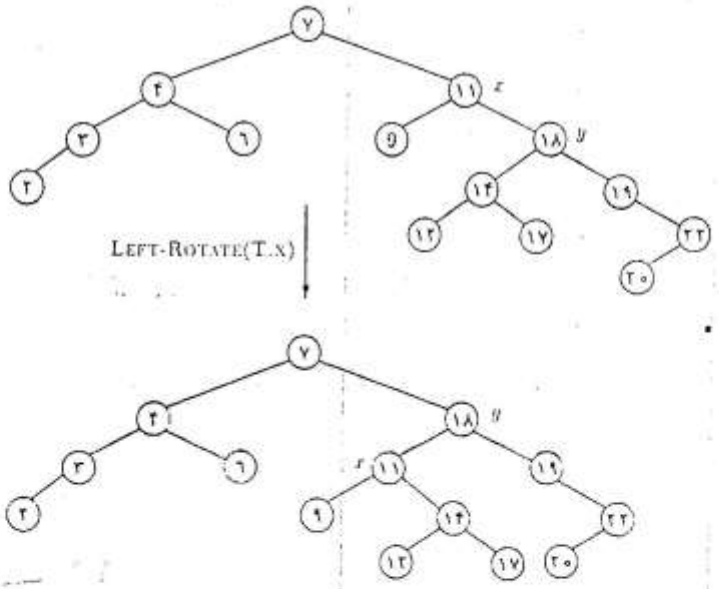
[1] Black Height

[2] رنگ x بی‌تاثیر است زیرا x را نمی‌شماریم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۷.۳: دوران راستگرد و چپگرد



شکل ۸.۳: مثالی از LEFT-ROTATE (T,x).

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

شبه دستورات لازم برای پیاده‌سازی دوران چپگرد در اینجا آورده شده است. دستورات لازم برای دوران راستگرد نیز شبیه به حالت چپگرد می‌باشد. واضح است که دو عمل فوق $O(1)$ می‌باشند زیرا فقط اشاره‌گرها در اثر یک دوران تغییر می‌کنند و سایر اجزا بدون تغییر می‌مانند.

```
Left-Rotate(T,x)
  y <- right[x]
  right[x] <- left[y]
  if left[y] <> NIL then p[left[y]] <- x
  p[y] <- p[x]
  if p[x] = NIL
  then Root[T] <- y
  else if x = left[p[x]]
          then left[p[x]] <- y
          else right [p[x]] <- y
  left[y] <- x
  p[x] <- y
```

شبه کد مربوط به Left Rotate

۳.۶.۳ درج

۱. براساس درج در درخت دودوئی جستجو، z را درج می‌نمائیم.

۲. z را قرمز می‌کنیم. بدیهی است که سیاه– ارتفاع‌ها برای کلیه گره‌ها ثابت می‌ماند.

۳. اگر پدر z سیاه باشد درج به پایان رسیده‌است.

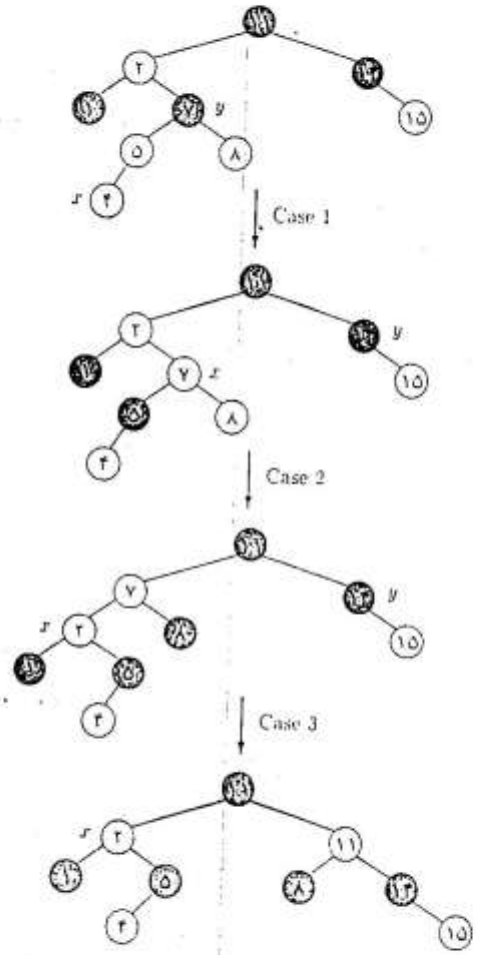
۴. اگر پدر z قرمز باشد به سراغ عموی z بنام y می‌رویم. (همانطور که در شبه دستورات مربوط به RB-Insert دیده می‌شود برحسب اینکه Parent[z] فرزند چپ یا فرزند راست جد z باشد، عموی z به ترتیب فرزند راست یا چپ جد z خواهد بود)

در قسمت بعد ایده اصلی این است که در هر مرحله با حفظ خاصیت ۴ مربوط به درخت قرمز– سیاه، اشکال موجود که مربوط به عدم برقراری خاصیت ۳ می‌باشد را برای ارتفاع فعلی درخت حل کنیم و مشکل را احیاناً به ارتفاعات بالاتر ارسال نمائیم و این کار را برای سطوح بالاتر آنقدر ادامه دهیم تا به ریشه برسیم یا شرط سوم (سیاه بودن پدر z) برقرار شود و اجرا خاتمه پذیرد. واضح است که درخت صرفاً زمانی نیاز به تصحیح دارد که پدر z قرمز باشد.

برحسب اینکه Parent[z] فرزند چپ یا راست جد z باشد، ۶ حالت متفاوت پیش می‌آید که ۳ حالت دیگر عیناً متقارن با سه حالت زیر می‌باشند. این حالات براساس رنگ عموی z، یعنی y تعیین می‌شوند.

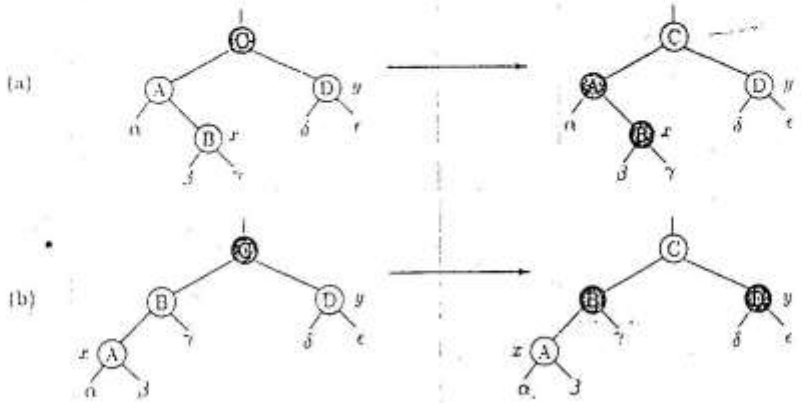
حالت اول: y قرمز است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

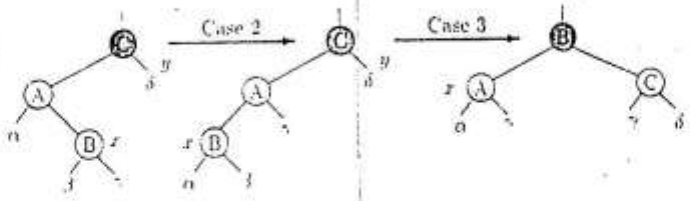


شکل ۹.۳: مثالی از درج.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۰.۳: حالت اول برای RB-INSERT.



شکل ۱۱.۳: حالت‌های دوم و سوم برای RB-INSERT.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

حالت دوم: y سیاه است و x فرزند راست پدرش است.

نکته: دیده می‌شود که در پایان اجرای RB-Insert برای حفظ خاصیت ۵، ریشه درخت سیاه می‌شود. شکل‌های ۱۰.۳ و ۱۱.۳ حالت‌های مختلف درج را نشان می‌دهند.

```
RB-Insert(T,x)
    Tree-Insert(T,x)
    color[x] <- Red
    while x <> root[T] and color[p[x]] = Red
      do if p[x] = left[p[p[x]]]
            then y <- right[p[p[x]]]
                 if color[y] = Red
                    then color[p[x]] <- Black                  .... case 1
                         color[y] <- Black                     .... case 1
                         color[p[p[x]]] <- Red                 .... case 1
                         x <- p[p[x]]                          .... case 1
                    else if x = right[p[x]]
                            then  x <- p[x]                    .... case 2
                                  Left-Rotate(T,x)             .... case 2
                         color[p[x]] <- Black                  .... case 3
                         color[p[p[x]]] <- Red                 .... case 3
                         Right-rotate(T,p[p[x]])               .... case 3
            else (same as then clause
                    with "right" and "left" exchanged)
    color[root[T]] <- Black
```

شبه کد مربوط به RB-Insert

۴.۶.۳ حذف

برای ساده سازی شرایط مرزی در پیاده سازی RB-Delete ناگزیریم که یک گرهٔ حفاظتی بنام nil[T] برای درخت T تعریف نمائیم. این گره حاوی همان مؤلفه‌هایی می‌باشد که در گره‌های معمولی درخت وجود دارد. مؤلفه رنگ این شیء سیاه می‌باشد و سایر مؤلفه‌ها می‌توانند مقادیر دلخواه بگیرند.

با استفاده از این شیء محافظتی می‌توانیم با یک فرزند nil مانند یک گره معمولی درخت که پدرش x می‌باشد رفتار نمائیم (استفاده مولفهٔ Parent مربوط به این شیء محافظتی در فراخوانی RB-Delete-Fixup و اولین اجرای حلقه While می‌باشد).

بنابراین باید در درخت قرمز– سیاه تمام اشاره‌گرهای nil را به nil[T] اشاره دهیم (در این مرحله، مؤلفه Parent متناظر با این شیء هنوز مقدار خاص و قابل استفاده‌ای در خود ندارد.)

برای حذف یک عنصر ابتدا بروشی مانند حذف از درخت دودویی جستجو، عنصر مورد نظر را حذف و سپس درخت را تصحیح می‌کنیم. باید دقت کنیم که شبه دستورات مربوط به حذف از درخت قرمز– سیاه با همتای خود در درخت جستجوی دودویی دقیقاً یکسان نیست چون در درخت قرمز– سیاه اشاره‌گرهای nil به nil[T] اشاره می‌کنند و

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

همچنین مؤلفه Parent[x] در خط ۷ بدون هیچ شرطی مقدار دهی می‌شود. زیرا در صورتی هم که x یک برگ nil باشد، nil[T].Parent مقدار خواهد گرفت. علاوه بر این دو تغییر، در صورتی که رنگ y سیاه باشد، درخت در خطوط ۱۶ و ۱۷ با فراخوانی RB-Delete-Fixup تصحیح می‌شود تا خاصیت ۴ در درخت قرمز– سیاه بازیابی شود.

```
    RB-Delete(T,z)
1.      if left[z] = nil[T] or right[z] = nil[T]
2.         then y <- z
3.         else y <- Tree-Successor(z)
4.      if left[y] <> nil[T]
5.         then x <- left[y]
6.         else x <- right[y]
7.      p[x] <- p[y]
8.      if p[y] = nil[T]
9.         then root[T] <- x
10.        else if y = left[p[y]]
11.                then left[p[y]] <- x
12.                else right[p[y]] <- x
13.     if y <> z
14.        then key[z] <- key[y]
15.              if y has other fields, copy them too.
16.     if color[y] = Black
17.        then RB-Delete-Fixup(T,x)
18.     return y
```

شبه کد مربوط به RB-DELETE

نکته: اگر RB-Delete با گره z بر روی درخت T فراخوانی شود با خود آن گره حذف می‌شود یا عنصر بعدی (Successor) آن حذف می‌شود. در حالت دوم دستور Key[z] := Key[Succ(z)] نیز اجرا خواهد شد.

قرارداد: گره حذف شده را y می‌نامیم.

اگر y سیاه باشد، حذف آن موجب می‌شود مسیری که در گذشته از y عبورمی‌کرده، یک گرهٔ سیاه کمتر از سایر مسیرها داشته باشد و این باعث از بین رفتن خاصیت ۴ در درخت قرمز– سیاه می‌شود. ما می‌توانیم این مشکل را با فرض اینکه بتوانیم گرهٔ x (که در حقیقت فرزند چپ یا راست گره y قبل از حذف شدن y بوده است) را یک بار اضافه‌تر سیاه کنیم، حل نمائیم. یعنی عبور از گره x به معنای عبور از دو گره سیاه باشد. بنابراین زمانی که گره y را حذف می‌کنیم رنگ آنرا به فرزندش یعنی x می‌فرستیم. تنها مشکل این است که امکان دارد x دوبار سیاه شده باشد (خودش از قبل سیاه بوده باشد) و این باعث از بین رفتن خاصیت ۱ در درخت قرمز– سیاه می‌شود.

هدف ما این است که این یک رنگ سیاه اضافی را به طرف ارتفاعات بالاتر در درخت بفرستیم تا به یکی از حالت‌های زیر برسیم:

الف) x به یک گره قرمز اشاره نماید که در این حالت ما آن گره را سیاه می‌کنیم.

ب) x به ریشه اشاره کند که در این حالت، سیاه اضافی را می‌توان دور انداخت.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

پ) با دورانهای مناسب و رنگ آمیزی مجدد بتوان خاصیت ۱ را احیاء نمود.

اگر در یک مرحله موفق نشویم که به یکی از سه وضعیت بالا برسیم ممکن است با ۴ حالت متفاوت روبرو شویم (در حقیقت برحسب اینکه x فرزند چپ یا راست باشد ۸ وضعیت دوبدو متقارن بوجود خواهد آمد.)

قراردادها:

- در هر مرحله x گره‌ای است که دو برچسب سیاه دارد.
- w برادر (sibling) گره x خواهد بود.

حالتهایی که ممکن است بوجود بیاید از قرار زیر است:

۱. w قرمز است (بنابراین فرزندان سیاه دارد.)

۲. w سیاه و هر دو فرزندش نیز سیاهند.

۳. w سیاه و فرزند چپش قرمز و فرزند راستش سیاه است.

۴. w سیاه و فرزند راستش قرمز است.

برای درک بهتر این مطلب، حالتهای مختلف در «شکل ۱۲.۳» آورده شده است.

قراردادها:

- گره‌های سیاه: پررنگ
- گره‌های قرمز: خاکستری
- گره‌های با رنگ نامعلوم (رنگ این گره‌ها با c و c' مشخص شده): سفید

حالت اول: حالت ۱ با تعویض رنگ گره‌های D و B با یکدیگر و انجام یک دوران چپگرد به یکی از حالتهای ۲ و ۳ یا ۴ تبدیل می‌شود.

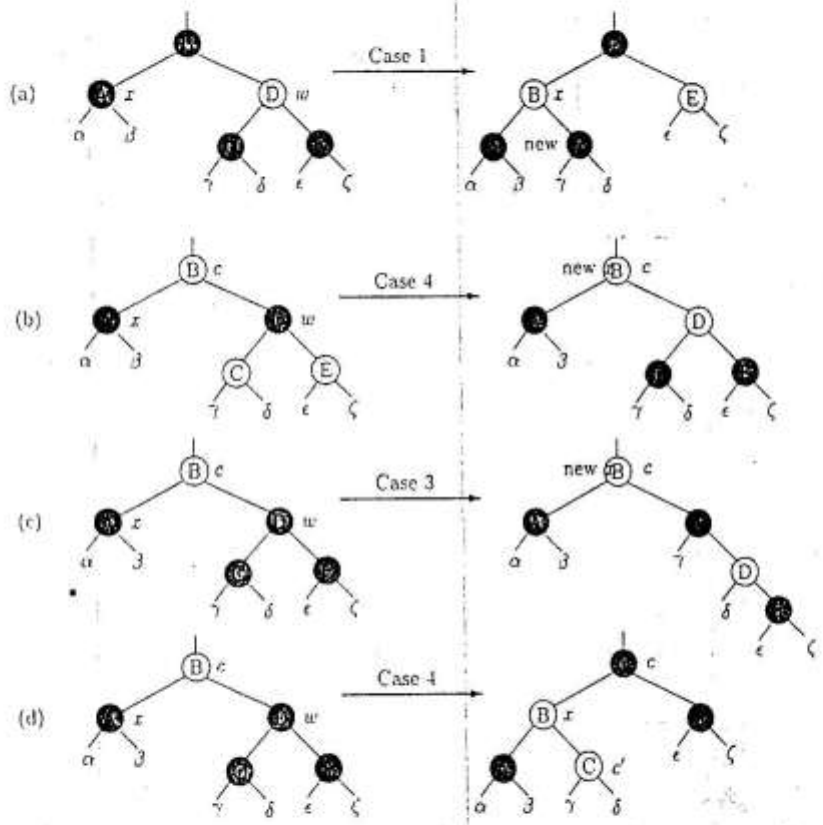
حالت دوم: رنگ سیاه اضافی که با اشاره‌گر x نمایش داده شده است با قرمز کردن D و اشاره دادن x به B به یک ارتفاع بالاتر منتقل می‌شود. اگر از طریق حالت اول وارد حالت ۲ شده باشیم چون c قرمز است حلقه به پایان خواهد رسید.

حالت سوم: حالت ۳ با تعویض رنگ گره‌های C و D و انجام یک دوران راستگرد به حالت ۴ تبدیل می‌شود.

حالت چهارم: در حالت ۴ رنگ سیاه اضافی که با x نمایش داده شده است را می‌توان با عوض کردن چند رنگ و یک دوران چپگرد بدون آنکه به خواص درخت لطمه‌ای وارد شود حذف نمود و حلقه در این مرحله به پایان می‌رسد.

واضح است که RB-Delete هم $O(\lg n)$ می‌باشد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۲.۳: حالتهای مختلف ایجاد شده در مراحل حذف

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
RB-Delete-Fixup(T,x)
while x <> rot[T] and color[x]=BLACK
    do if x = left[p[x]]
          then w <- right[p[x]]
               if color[w] = RED
                  then color [w] <- BLACK                          .... Case 1
                       color[p[x]] <- RED                          .... Case 1
                       Left-Rotate (T, p[x])                       .... Case 1
                       w <- right [p[x]]                           .... Case 1
               if color[left[w]]=BLACK and color[right[w]]=BLACK
                  then color[w] <- RED                             .... Case 2
                       x <- p[x]                                   .... Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] <- BLACK             .... Case 3
                               color[w] <- RED                     .... Case 3
                               Right-Rotate(T,w)                   .... Case 3
                               w <- right[p[x]]                    .... Case 3
                       color[w] <- color[p[x]]                     .... Case 4
                       color[p[x]] <- BLACK                        .... Case 4
                       color[right[w]] <- BLACK                    .... Case 4
                       Left-Rotate(T,p[x])                         .... Case 4
                       x <- root[T]                                .... Case 4
          else (same as then with right and left exchanges)
color[x] <- BLACK
```

شبه کد مربوط به RB-DELETE-FIXUP

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

فصل ۴

روش‌های طراحی الگوریتم‌ها

روش‌های طراحی الگوریتم‌ها به منظور حل مسئله‌های مختلف را می‌توان به صورت زیر تقسیم‌بندی کرد:

۱. مبتنی بر استقرا[1]

۲. تقسیم و حل[2]

۳. برنامه‌ریزی پویا[3]

۴. حریصانه[4]

۵. جستجوی فضای حالت[5]

روش پس‌گرد[6] جستجوی فضای حالت را به نظم در می‌آورد. برای محدود کردن فضای جستجو، که اغلب بسیار بزرگ است، از «درخت بازی[7]»، به خصوص هرس $\alpha - \beta$ و نیز روش «انشعاب و تحدید[8]» استفاده می‌شود. در صورتی که راه حل سریع بسیار کند باشد، می‌توان از روش‌های تقریبی یا مکاشفه‌ای[9] استفاده کرد.

۱.۴ استقرای ریاضی

اثبات حکمی برای حالت کلی n به روش استقرای ساده شامل مرحله‌های زیراست:

پایه‌ی استقرا[10]: ثابت می‌کنیم حکم برای $n = 1$ درست است.

فرض استقرا[11]: فرض می‌کنیم حکم برای $n - 1$ درست است.

---

[1] induction
[2] divide and conquer
[3] dynamic programming
[4] greedy
[5] state space search
[6] backtracking
[7] game tree
[8] branch and bound
[9] heuristic

حکم استقرا[۱۲]: ثابت می‌کنیم حکم برای n درست است.

البته استقرای قوی[۱۳] هم داریم که در آن در فرض استقرا، فرض می‌کنیم حکم برای $i < n$ درست است، و حکم را برای برای $i = n$ ثابت می‌کنیم. برای مثال:

پایه: حکم برای $n = ۲$ و $n = ۱$ درست است.

فرض : حکم برای $i > ۲$ درست است.

حکم : برای $i + ۲$ درست است.

چون ثابت کردیم حکم برای $n = ۱$ و $n = ۲$ و $n + ۲$ درست است پس ثابت می‌شود که برای تمام اعداد طبیعی هم درست است. و در مثالی دیگر:

پایه: برای $n = ۱$ درست است.

فرض : برای $n = ۲^k$ درست است.

حکم : برای $n = ۲^{k+۱}$ نیز درست است.

بدین ترتیب ثابت می شود که برای کلیه‌ی توانهای ۲ درست است.

مثال ۱ — محاسبه‌ی فرمول

مقدار $T(n)$ را به صورت فرمولی برحسب n محاسبه کنید:

$$T(n) = ۸ + ۱۳ + ۱۸ + ۲۳ + \cdots + (۳ + ۵n)$$

حدس می زنیم که $T(n) = c_۱n^۲ + c_۲n + c_۳$. برای اثبات، ابتدا مقادیر $c_۱$، $c_۲$ و $c_۳$ را به دست می آوریم و به کمک استقرا ثابت می کنیم که این دو رابطه هم‌ارزند.

$$\begin{aligned}
T(۱) &= ۸ \Rightarrow c_۱ + c_۲ + c_۳ = ۸ \\
T(۲) &= ۲۱ \Rightarrow ۴c_۱ + ۲c_۲ + c_۳ = ۲۱ \\
T(۳) &= ۳۹ \Rightarrow ۹c_۱ + ۳c_۲ + c_۳ = ۳۹
\end{aligned}$$

بنابراین:

$$\begin{aligned}
c_۱ &= \frac{۵}{۲} \\
c_۲ &= \frac{۱۱}{۲} \\
c_۳ &= ۰
\end{aligned}$$

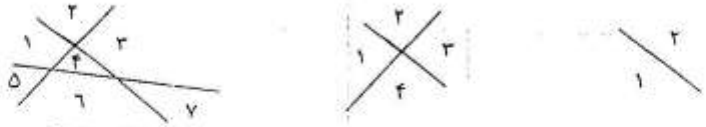
یعنی باید ثابت کنیم:

$$T(n) = \frac{۵}{۲}n^۲ + \frac{۱۱}{۲}n$$

حال به کمک استقرا حکم بالا را ثابت می کنیم.

[۱۲] strong induction

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



الف) $S(۱) = ۲$ ب) $S(۲) = ۴$ ج) $S(۳) = ۷$

شکل ۱.۴: شمارش ناحیه‌های بین خط‌ها در یک صفحه

پایه : $T(۱) = ۸$

فرض : $T(n-۱) = \frac{۵}{۲}(n-۱)^۲ + \frac{۱۱}{۲}(n-۱)$

حکم : $T(n) = \frac{۵}{۲}n^۲ + \frac{۱۱}{۲}n^۲$

$$\begin{aligned} T(n) &= T(n-۱) + ۳ + ۵n \\ &= \frac{۵}{۲}(n-۱)^۲ + \frac{۱۱}{۲}(n-۱) + ۳ + ۵n \\ &= \frac{۵}{۲}(n^۲ - ۲n + ۱) + \frac{۱۱}{۲}(n-۱) + ۳ + ۵n \\ &= \frac{۵}{۲}n^۲ - ۵n + \frac{۵}{۲} + \frac{۱۱}{۲}n - \frac{۱۱}{۲} + ۳ + ۵n \\ &= \frac{۵}{۲}n^۲ + \frac{۱۱}{۲}n \end{aligned}$$

مثال ۲ — شمارش تعداد ناحیه‌های بین n خط در صفحه

n خط در صفحه‌ای نامتناهی داده شده‌اند. با این فرض که خط‌ها دوبدو ناموازی هستند و هم‌چنین هیچ سه خط در یک نقطه یکدیگر را قطع نمی کنند، می خواهیم تعداد ناحیه‌های بین n خط را محاسبه کنیم.

برای حل می توانیم حدس بزنیم که با رسم خط nام چند ناحیه اضافه می شود.

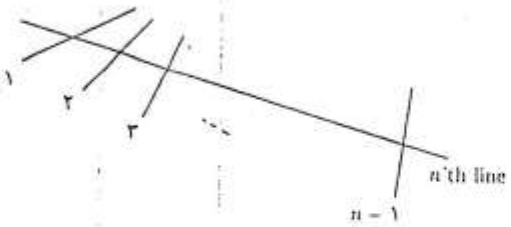
$$S(۱) = ۲$$

$$S(n) - S(n-۱) = ?$$

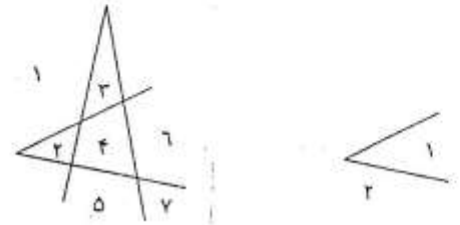
فرض می کنیم تعداد ناحیه‌های ایجاد شده با $n-۱$ خط $S(n-۱)$ باشد. وقتی خط nام را رسم می کنیم چون موازی با هیچ‌یک از خط‌ها نیست پس هر خط را در یک نقطه قطع می کند. روی خط nام $n-۱$ نقطه‌ی تلاقی با $n-۱$ خط قبلی خواهیم داشت (شکل ۲.۴).

هیچ‌یک از نقاط تلاقی با خط nام تکراری نیست چون هیچ سه خطی یکدیگر را در یک نقطه قطع نمی‌کنند. بنابراین خط nام به n پاره‌خط تقسیم می‌شود. این پاره‌خط‌ها هرکدام از درون یکی از ناحیه‌های قبلی می‌گذرد و لذا آن ناحیه را به دو ناحیه‌ی جدید تقسیم می کند. پس n ناحیه به ناحیه‌هایی که با $n-۱$ خط ایجاد شده‌بود اضافه

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲.۴: تلاقی خط nام با بقیه خط‌ها.



الف) $S(۱) = ۲$ ب) $S(۲) = ۷$

شکل ۳.۴: شمارش ناحیه‌های بین زاویه‌ها در یک صفحه

می شود.

$$\begin{aligned} S(n) &= S(n-۱) + n \\ S(۱) &= ۲ \end{aligned}$$

$$\begin{aligned} S(n) &= S(n-۱) + n - ۱ + n \\ &= S(n-۲) + n - ۲ + n - ۱ + n \\ &= S(۱) + ۲ + ۳ + \ldots + n \\ &= ۲ + ۲ + ۳ + \ldots + n \\ S(n) &= \frac{n(n-۱)}{۲} + ۱ \end{aligned}$$

مثال ۳– شمارش حداکثر تعداد ناحیه‌های بین n زاویه

می‌خواهیم حداکثر تعداد نواحی ایجاد شده با رسم n زاویه (یا زاویه‌های دلخواه) در یک صفحه را به دست آوریم.

پایه‌ی استقرا: $D(۱) = ۲$ بدیهی است.

فرض استقرا: حداکثر تعداد نواحی ایجاد شده با $n - ۱$ زاویه برابر است با $D(n - ۱)$. زاویه‌ی nام را چنان رسم می کنیم که هر ضلع آن هر دو ضلع تمامی زاویه‌های قبلی را قطع کند. برای این کار راس این زاویه باید در

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

یک ناحیه‌ی خارجی انتخاب شود. یعنی ناحیه‌ای که داخل هیچ زاویه‌ای قرار نداشته باشد. به عنوان مثال ناحیه‌ی ۲ در شکل ۳.۴.الف و ناحیه‌های ۶ و ۷ در شکل‌شکل ۳.۴.ب خارجی هستند.

حکم استقرا: برای محاسبه‌ی $D(n)$ کافیست مقدار d یعنی تفاضل $D(n)$ و $D(n-1)$ را پیدا کنیم. برای محاسبه‌ی d توجه می کنیم که هنگام رسم ضلع اول زاویه‌ی nام تعداد نیم‌خط‌های موجود در صفحه برابر است با $2(n-1)$. پس می‌توانیم حداکثر $2(n-1)$ نقطه‌ی تلاقی با خط جدید ایجاد کنیم. در این صورت تعداد ناحیه‌های اضافه شده $2(n-1)+1$ خواهد بود. اکنون تعداد نیم‌خط‌های موجود در صفحه برابر است با $2n-1$. بنابراین با رسم ضلع دوم زاویه $2n$ ناحیه‌ی دیگر اضافه می شود. اما همان طور که در شکل دیده می‌شود به دلیل این که امتداد دو ضلع وجود ندارد، دو ناحیه از مجموع کل ناحیه‌ها کسر می شود. در نتیجه داریم $d=4n-3$. حال با باز کردن رابطه‌ی بازگشتی، فرمول دلخواه را به دست می آوریم:

$$
\begin{aligned}
D(n) &= D(n-1)+4n-3\\
&= D(n-2)+4[n+(n-1)]-2\cdot 3\\
&= D(n-3)+4[n+(n-1)+(n-2)]-3\cdot 3\\
&\vdots\\
&= D(n-i)+4[n+(n-1)+(n-2)+\dots+(n-i+1)]-i\cdot 3\\
&= D(1)+4[n+(n-1)+(n-2)+\dots+3+2]-(n-1)\cdot 3\\
&= 2+4(n+2)(n-1)/2-3(n-1)\\
&= 2n^2-n+1
\end{aligned}
$$

مثال ۴– رنگ کردن ناحیه‌های بین خط‌ها

یک صفحه با n خط در آن داده شده است. حداقل با چند رنگ می تواند ناحیه‌های ایجاد شده را رنگ کرد به طوری که هیچ دو ناحیه‌ی مجاوری یک‌رنگ نشوند؟

پایه: اگر یک پاره‌خط داشته باشیم یک طرف آن را با رنگ ۱ و طرف دیگر را با رنگ ۲ رنگ می کنیم.

فرض : $n-1$ خط داریم و با دو رنگ ناحیه‌های ایجاد شده‌ی بین آن‌ها را رنگ کرده‌ایم.

حکم : خط nام را را رسم می‌کنیم (این خط می تواند با یک یا چند خط موازی باشد.) این خط صفحه را به دو قسمت a و b تقسیم می‌کند. این خط نیز از میان تعدادی ناحیه می‌گذرد و هرکدام را به دو ناحیه، یکی در قسمت a و دیگری در قسمت b، تقسیم می‌کند. کافی است تا رنگ کلیه‌ی ناحیه‌های یک قسمت، چه جدید و چه قدیم، را به رنگ متضاد آن‌ها تغییر دهیم و رنگ‌های ناحیه‌های قسمت دیگر را تغییر ندهیم. بدیهی‌است که رنگ‌های ناحیه‌های مجاور در هر قسمت پس از این عمل متضاد خواهند بود (با توجه به فرض استقرا.) تنها می‌مانند زوج‌ناحیه‌های مجاوری که پاره‌خط مشترکشان قسمتی از خط nام است. این ناحیه‌ها قبلاً یک ناحیه بودند و دارای یک رنگ؛ با تغییر رنگ همه‌ی ناحیه‌های یک قسمت، حتماً رنگ‌های این زوج‌ناحیه‌ها هم متضاد خواهند بود.

مثال ۵

هرم زیر موجود است:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

1	1
3 + 5	8
7 + 9 + 11	27
13 + 15 + 17 + 19	64
21 + 23 + 25 + 27 + 29	125
⋮	

مجموع اعداد سطر i ام را بدست آورید.

حدس : مجموع اعداد سطر i ام = i^3 $\qquad T(i) = i^3$

فرض :

$$\begin{aligned}
T(1) &= 1 \\
T(i) &= a_1 + a_2 + a_3 + a_4 + \cdots + a_i \\
T(i+1) &= b_1 + b_2 + b_3 + b_4 + \cdots + b_i + b_{i+1} \\
b_1 &= a_2 + 2 \\
a_2 &= a_1 + 2
\end{aligned}$$

$$\begin{aligned}
b_1 - a_1 &= 2i \\
b_2 - a_2 &= 2i \\
&\vdots \\
b_i - a_i &= 2i
\end{aligned}$$

$$T(i+1) = T(i) + (2i) \times i + b_{i+1}$$

b_{i+1} برابر با $k = (i+1)(i+2)/2$ امین عدد فرد است، یعنی $k = 1 + 2 + 3 + 4 + \cdots + i + 1$ پس:

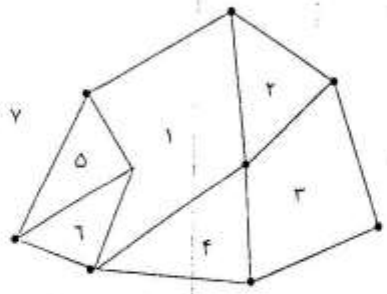
$$b_{i+1} = 2\frac{(i+1)(i+2)}{2} - 1 = i^2 + 3i + 1$$

$$\begin{aligned}
T(i+1) &= T(i) + 2i^2 + i^2 + 3i + 1 \\
&= i^3 + 2i^2 + 3i + 1 \\
&= (i+1)^3
\end{aligned}$$

مثال ۶ — فرمول اولر در گراف‌های مسطح و همبند

در ابتدا به تعریف‌های زیر می‌پردازیم:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴.۴: یک گراف مسطح با $F = ۷$، $E = ۱۴$، و $V = ۹$. داریم: $۹ + ۷ = ۱۴ + ۲$

راس منفرد: راسی که هیچ یالی به آن متصل نباشد.

گراف همبند[۱۴]: گرافی که از هر راس آن به هر راس دیگری لااقل یک مسیر وجود داشته باشد.

گراف مسطح[۱۵]: گرافی که بتوان آن را در صفحه به طوری رسم کرد که یال‌های آن همدیگر را قطع نکنند.

درخت آزاد[۱۶]: گراف همبند بدون جهت و بدون دور.

مساله – گراف همبند و مسطح با V راس، E اتصال و F ناحیه داده شده است ثابت کنید: $V + F = E + ۲$.
(به مثال شکل ۴.۴ توجه کنید.)

استقرا را روی F انجام می دهیم:

پایه‌ی استقرا: $F = ۱$: گراف یک درخت آزاد است.

قضیه ۵. در هر درخت آزاد با V راس و E یال داریم $V = E + ۱$.

اثبات. اگر درخت را ریشه‌دار فرض کنیم:

در این صورت به هر راسی، به جز ریشه، یک یال وارد شده است، پس تعداد رأس‌ها یکی بیش از تعداد یال‌هاست. اما برای درخت آزاد در حالت کلی دوباره از استقرا استفاده می‌کنیم (به این کار به اصطلاح استقرای دوبل[۱۷] گفته می‌شود.)

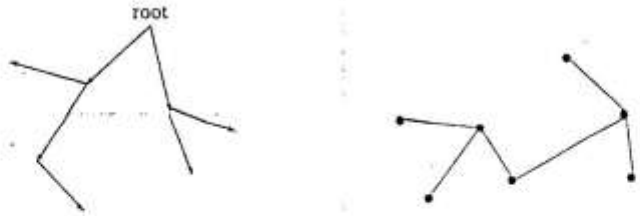
- روش اول: استقرا روی V:

پایه‌ی استقرا: $V = ۲, E = ۱$ بدیهی است.

فرض استقرا: برای $V - ۱$ رابطه برقرار است.

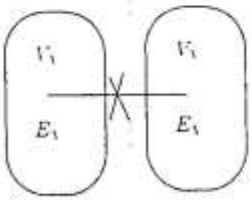
[۱۴] connected graph
[۱۵] planar graph
[۱۶] free tree
[۱۷] double induction

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



الف) درخت آزاد ب) درخت ریشه‌دار

شکل ۵.۴: فرمول اولر برای درخت‌ها.



شکل ۶.۴: اثبات فرمول اولر برای درخت آزاد.

حکم استقرا: برای V رابطه برقرار است. حتما یک راس با درجه‌ی ۱ وجود دارد زیرا در غیر این صورت دور ایجاد می‌شود. این راس و یال متصل به آن را حذف می‌کنیم. طبق فرض استقرا در گراف حاصل رابطه‌ی $V - 1 = (E - 1) + 1$ برقرار است. با افزودن یک واحد به طرفین معادله داریم: $V = E + 1$.

- روش دوم: استقرا روی E:

پایه‌ی استقرا: $V = 1, E = 0$ بدیهی است.

فرض استقرا: برای $E' < E$ رابطه برقرار است. (توجه شود در این جا از استقرا قوی استفاده می کنیم.)

حکم استقرا: برای E رابطه برقرار میباشد. یک یال دلخواه را حذف می کنیم با توجه به این که در درخت دور وجود ندارد هیچ مسیر دیگری بین دو رأس دو سر این یال وجود نخواهد داشت. بنابراین درخت به دو درخت با تعداد یال‌های کمتر تجزیه می‌شود. فرض کنید تعداد رأس‌ها و یال‌های این دو درخت به ترتیب V_1، V_2، E_1 و E_2 باشد.

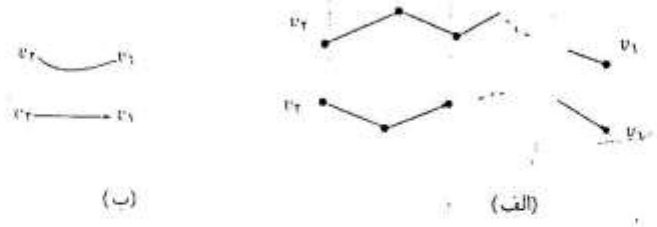
طبق فرض استقرا داریم: $V_1 = E_1 + 1$ و $V_2 = E_2 + 1$. از جمع این دو رابطه داریم: $V_1 + V_2 = (E_1 + E_2 + 1) + 1$. که با در نظر داشتن یال حذف شده به دست می آوریم:

$$\boxed{V = E + 1}$$

بنابراین حکم برای $F = 1$ ثابت شده است. □

فرض استقرای مساله‌ی اصلی : رابطه برای $F - 1$ برقرار است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۷.۴: v_1 و v_2 مجاور هم هستند.

حکم استقرا: رابطه برای F نیز برقرار می باشد.

یک ناحیه وجود دارد که همسایه‌ی ناحیه‌ی خارجی است. این ناحیه در داخل یک دور محاط شده است. حال یک یال که مرز این ناحیه و ناحیه‌ی خارجی باشد را حذف می کنیم. از آن‌جا که این یال از یک دور حذف شده گراف حاصل هم‌چنان همبند باقی خواهد ماند. با انجام عمل فوق ناحیه‌ی مزبور در ناحیه‌ی خارجی ادغام می شود. پس طبق فرض استقرا در این گراف جدید داریم: $V + (F - 1) = (E - 1) + 2$. با افزودن یک واحد به طرفین معادله‌ی فوق به دست می آوریم: $V + F = E + 2$، و حکم ثابت است.

مثال ۷– مجموعه‌ی مستقل در گراف‌ها

در ابتدا به تعریف‌های زیر می‌پردازیم:

دسترسی[18]: رأس v_1 توسط رأس v_2 قابل دسترسی است اگر مسیری از v_2 به v_1 وجود داشته باشد.

مجاورت[19]: راس v_1 مجاور راس v_2 است اگر یالی از v_2 به v_1 موجود باشد. توجه کنید که روابط فوق در یک گراف بدون جهت خاصیت تقارنی دارند (شکل ۷.۴).

زیرگراف القایی[20]: در گراف $G = (V, E)$ زیرگراف القایی G_1 بامجموعه‌ی رأس‌های H گرافی‌است که مجموعه‌ی رأس‌های آن H و مجموعه‌ی یال‌های آن $H \times H \cap E(g)$ است، یعنی همه‌ی یال‌هایی از G که دو سرشان در H است (شکل ۸.۴).

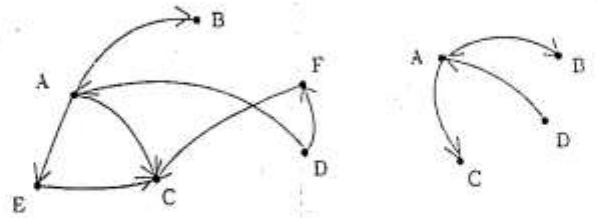
مجموعه‌ی مستقل[21] در یک گراف: در گراف $G = (V, E)$ زیرمجموعه‌ای از V مثل H یک مجموعه‌ی مستقل است اگر هیچ دو رأس موجود در H با هم مجاور نباشند. به عنوان مثال مجموعه‌ی $\{B, E, F\}$ در شکل ۸.۴ یک مجموعه‌ی مستقل است.

مجموعه‌ی همسایگی[22]: در گراف $G = (V, E)$ مجموعه‌ی همسایگی یک رأس مثل v به صورت زیر تعریف می شود:

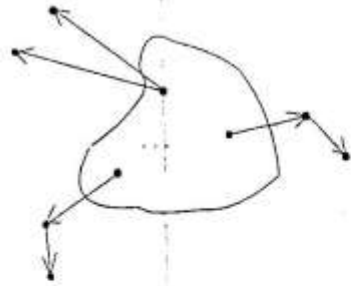
$$N(v) = \{v\} \cup \{w \in V | (v, w) \in E\}$$

[18] reachability
[19] adjacency
[20] induced subgraph
[21] independent set
[22] neighborhood

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

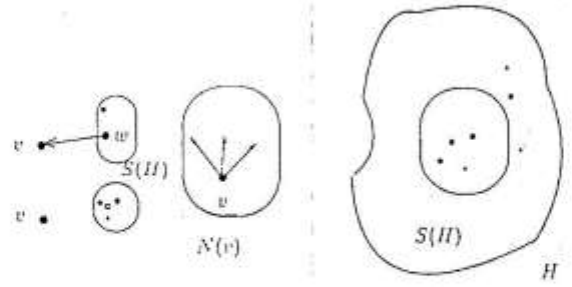


شکل ۸.۴: زیرگراف القایی G_1 با مجموعه‌ی رأس‌های $\{A, B, C, D\}$ از گراف G



شکل ۹.۴: مجموعه‌ی مستقل.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۰.۴: اثبات قضیه.

یعنی مجموعه‌ای شامل خود v و تمام رأس‌های مجاورش.

قضیه ۶. برای هر گراف جهت دار $G = (V, E)$ یک مجموعه‌ی مستقل $S(G)$ وجود دارد که هر رأس گراف از یکی از رأس‌های این مجموعه‌ی مستقل با مسیری به طول حداکثر ۲ قابل دسترسی است.

اثبات. بر روی n (تعداد رأس‌ها) استقرا انجام می‌دهیم.

پایه‌ی استقرا: برای $n <= ۳$ بدیهی است.

فرض استقرا: برای $n' < n$ قضیه درست است. (از استقرای قوی استفاده می کنیم.)

حکم استقرا: برای n نیز درست است.

راس v و زیرگراف القایی H با مجموعه‌ی رأس‌های $V - N(v)$ را در نظر بگیرید. طبق فرض استقرا H دارای مجموعه‌ی مستقلی مثل $S(H)$ است که خاصیت مطلوب را دارا می‌باشد. حالا دو حالت زیر را بررسی می‌کنیم:

حالت اول — راس v مجاور یکی از رأس‌های $S(H)$ مثل w است:

در این صورت همان مجموعه‌ی $S(H)$ جواب است ($S(G) = S(H)$.) زیرا همه‌ی رأس‌های موجود در $N(v)$ از w با مسیری به طول حداکثر ۲ قابل دسترسی هستند.

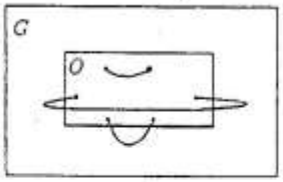
حالت دوم — راس v مجاور هیچ کدام از رأس‌های $S(H)$ نیست:

در این صورت با توجه به این که هیچ یک از رأس‌های $S(H)$ نیز مجاور v نیست می‌توان v را به $S(H)$ اضافه کرد و مجموعه‌ی حاصل هم‌چنان مستقل خواهد بود. و مابقی رأس‌های موجود در $N(v)$ اکنون از v با مسیری به طول ۱ قابل دسترسی خواهند بود. بنابراین $S(G) = S(H) \cup \{v\}$ مجموعه‌ی خواسته شده است.

برای مثال در شکل ۸.۴ می توانیم $S(G)$ را برابر مجموعه‌ی D بگیریم.

□

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۱.۴: مسیرهای مستقل یالی.

یک مسئله با استفاده از قضیه‌ی فوق

الگوریتمی بنویسید که گره‌هایی را در یک گراف داده شده رنگ کند به طوری که هیچ یک مجاور دیگری نبوده و فاصله‌ی هر گره گراف از یکی از آن‌ها کمتر از ۳ باشد.

روش حل: یک گره‌ی دلخواه را در نظر گرفته و آن گره و کلیه‌ی رأس‌های مجاورش را از گراف حذف می کنیم. به صورت بازگشتی مسئله را برای گراف کوچکتر حاصل حل می کنیم. ختم بازگشت وقتی است که تعداد رأس‌ها کمتر از ۲ شود. حال اگر گره‌ی حذف شده مجاور هیچ یک از رأس‌های گراف جدید نباشد آن را نیز رنگ می کنیم.

گراف را می توان به عنوان یک شهر و رأس‌های رنگ شده را به عنوان مراکز آتش نشانی در نظر گرفت.

مثال ۸ – مسیرهای مستقل یالی

تعریف مسیرهای مستقل یالی[۲۲]: دو مسیر در یک گراف مستقل یالی هستند اگر دارای یال مشترک نباشند.

قضیه ۷. در گراف $G = (V, E)$ فرض کنید O مجموعه‌ی رأس‌های با درجه‌ی فرد باشد. O را می‌توان به زوج‌راس‌هایی تبدیل کرد که مسیرهای بین این زوج‌راس‌ها مستقل یالی باشند.

شکل ۱۱.۴ مثالی از این قضیه است.

اثبات. در مجموعه‌ی فوق (O) رابطه‌ی زیر برقرار است:

$$|O| = 2k$$

ابتدا رابطه‌ی فوق را اثبات می‌کنیم. می‌دانیم که $\Sigma_{v_i \in V} deg(v_i)$ یک عدد زوج است چون برای هر یال موجود در گراف، دو درجه به این مجموع اضافه می‌کند. از طرف دیگر هر راس موجود در مجموعه‌ی $V - O$ از درجه‌ی زوج است بنابراین $\Sigma_{w_i \in V-O} deg(w_i)$ عدد زوجی است. چون:

$$\Sigma_{v_i \in V} deg(v_i) = \Sigma_{w_i \in V-O} deg(w_i) + \Sigma_{u_i \in O} deg(u_i)$$

چون برای هر $(u_i \in O)$، $deg(u_i)$ عدد فردی است بنابراین تعداد u_i ها باید زوج باشد یعنی رابطه‌ی فوق الذکر صحیح است.

[۲۲] Edge Disjoint

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

شکل ۱۲.۴: اثبات غلط قضیه‌ی اولر.

حال با استقرا روی تعداد یال‌ها قضیه را ثابت می کنیم؛ ابتدا فرض می‌کنیم که گراف همبند نیست. بنابراین گراف مورد نظر ما به چند گراف همبند کوچکتر تقسیم می‌شود که برای هر گراف همبند قضیه صادق است (بنا به فرض استقرا.) بنابراین برای کل گراف نیز صحیح است.

حال گراف G را در نظر بگیرید که u و v دو عضو O باشند و همبند باشد مسیری از v به u وجود دارد که با حذف این مسیر ممکن است G به یک گراف غیر همبند تبدیل شود که در آن گراف مجموعه‌ی رئوس با درجه‌ی فرد برابر است با: $O - \{u, v\}$ که با فرض استقرا قضیه برای این گراف صادق است. در غیر این صورت با حذف این مسیر گراف همبند می ماند یعنی برای هر دو راس مسیری بین آن دو هست که باز می توان عمل فوق را تکرار کرد. □

۱.۱.۴ خطاهای معمول در اثبات با استقرا

به چند نوع خطاهایی که ممکن است در استفاده از استقرا ایجاد شود توجه کنید:

۱. در اثبات فرمول اولر ($V + F = E + ۲$) شخصی این قضیه را این چنین اثبات می‌کند:

اثبات به طریقه استقرا: یک وجه داریم، از وسط یک یال به وسط یال دیگری یک یال وصل می‌کنیم (شکل ۱۲.۴.)

V دو واحد اضافه شده،

F یک واحد اضافه شده، و

E سه واحد اضافه شده است.

پس رابطه‌ی مزبور صحیح است.

اشکال این اثبات در این‌جاست که اگر از پایه‌ی استقرا شروع کنیم و با استفاده از حکم مرحله به مرحله گراف را بزرگ‌تر کنیم، ممکن است گراف داده شده تولید نشود، چرا که راسی که اضافه می‌شود همیشه از درجه‌ی فرد است.

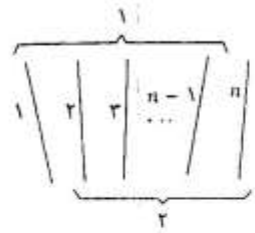
۲. n خط در صفحه داریم ثابت می‌کنیم که همگی از یک نقطه می‌گذرند.

اثبات به طریقه استقرا:

پایه: برای $n = ۲$ مسئله صحیح است.

فرض: برای $n - ۱$ خط مسئله صحیح است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۳.۴: اثبات غلط با استقرا.

حکم: برای n خط مسئله صحیح است.

مطابق شکل ۱۲.۴، $n - ۱$ خط شماره‌ی I بنا به فرض از یک نقطه می‌گذرند. $n - ۱$ خط شماره‌ی II هم بنا به فرض از یک نقطه می‌گذرند. این دو نقطه باید یکی باشند چون این دو دسته خط شامل خط‌های مشترکند. بنابراین کل n خط از یک نقطه می‌گذرند.

اشکال اثبات در پایه‌ی استقرا است، پایه باید $n = ۳$ باشد، نه $n = ۲$ و برای $n = ۳$، پایه درست نیست!

۳. ثابت کنید که

$$n = \sqrt{۱ + (n - ۱)\sqrt{۱ + (n)\sqrt{۱ + \cdots}}}$$

اثبات به وسیله‌ی استقرا :

پایه : برای $n = ۱$ مسئله صحیح است.

طرفین فرمول فوق را به توان دو می‌رسانیم، داریم:

$$n^۲ - ۱ = (n - ۱)\sqrt{۱ + (n)\sqrt{\cdots}} \Rightarrow n + ۱ = \sqrt{۱ + (n)\sqrt{\cdots}}$$

که همان فرمول فوق برای $n + ۱$ است و مسئله اثبات شده است.

اشکال اثبات در این‌جاست که ما بر $n - ۱$ تقسیم کرده‌ایم درحالی که پایه‌ی استقرا $n = ۱$ است؛ یعنی که در همان مرحله‌ی اول ما بر صفر تقسیم کرده‌ایم که اشکال دارد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۲.۴ طراحی الگوریتم با استقرا

برای حل مسایلی که ماهیت بازگشتی دارند، استقرا یک روش بسیار قوی است. به کمک استقرا می‌توان برای چنین مسایلی الگوریتم مناسب طراحی کرد، درستی الگوریتم را اثبات و آن را تحلیل کرد. در این جا برای چند مسئله‌ی این مرحله‌ها را نشان می‌دهیم.

۱.۲.۴ کُد گری

می‌خواهیم به n شئ کدهای متمایزی اختصاص دهیم. ساده‌ترین روش برای این کار آن است که به آنها اعداد ۱ تا n اختصاص دهیم و با توجه به آن که تمام کارهای ما در سیستم دودویی انجام می‌گیرد این n عدد را به صورت دودویی نمایش دهیم. بدیهی است که در این حالت تعداد بیت‌ها برای هر کد برابر است با $\lceil \log_2 n \rceil$ که به‌ترین حالت است.

ولی به علت محدودیت‌های سخت‌افزاری در شمارنده‌ها می‌خواهیم که کدهای متوالی فقط در یک بیت اختلاف داشته باشد. یکی از این دلایل این محدودیت جلوگیری از ایجاد Hazard در مدارهای منطقی‌ای است که این کدها را پشت سر هم می‌شمارد.

بنابراین مسئله که می‌خواهیم حل کنیم این است: چگونه می‌توان به n شئ کد دودویی با تعداد بیت کمینه (یعنی $\lceil \log_2 n \rceil$ بیت) اختصاص داد تا کدهای متوالی فقط در یک بیت اختلاف داشته باشند؟ توجه کنید که این توالی برای کدها «دورانی» است؛ یعنی هر کد دقیقاً دو کد قبل و بعد دارد. به چنین کدی «کد گری[۲۴]» می‌گویند.

کدهای گری دو نوع هستند؛ بسته و باز.

کد گری بسته[۲۵]: هرگاه هر کد با هرکدام از دو کد بعدی و قبلی‌اش دقیقاً در یک بیت اختلاف داشته باشد. بنابراین کدها تشکیل یک دور می‌دهند. مثلاً، {۰۰۰۰ → ۰۰۰۱ → ۰۰۱۱ → ۰۱۱۱ → ۰۱۱۰ → ۱۱۱۰ → ۱۰۱۰ → ۱۰۱۱ → ۱۱۱۱ → ۱۰۰۰} یک کد بسته برای ۱۰ عنصر است. توجه کنید که ۰۰۰۰ و ۱۰۰۰ هم در یک بیت با هم اختلاف دارند.

کد گری باز[۲۶]: همان کد بسته است مگر بین تنها دو کد متوالی که فقط اختلاف آن دو بیش از یک بیت است. مثلاً، {۰۰۰۰ → ۰۰۱۰ → ۰۰۱۱ → ۰۱۱۱ → ۰۱۱۰ → ۱۱۱۰ → ۱۱۱۱ → ۱۰۱۱ → ۱۰۱۰} یک کد باز برای ۹ عنصر است.

حالت‌های ساده‌ی زیر را در نظر بگیرید:

۱. برای $n = ۲$ کد بسته‌ی بهینه داریم: {۰ → ۱}

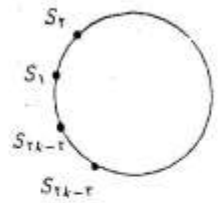
۲. برای $n = ۴$ نیز کد بسته‌ی بهینه داریم: {۰۰ → ۰۱ → ۱۱ → ۱۰}

۳. برای $n = ۳$ کد بسته نداریم ولی کد باز بهینه داریم: {۰۰ → ۰۱ → ۱۱}

لم ۴. ثابت کنید که برای تعداد زوجی عناصر ($n = ۲k$) می‌توان کد گری بسته ایجاد کرد.

[۲۴] Gray code
[۲۵] close Gray code
[۲۶] Open Gray Code

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۴.۴: نمایش کد گری

اثبات. این لم را به کمک استقرا به صورت «سازنده[17]» اثبات می‌کنیم؛ یعنی هم اثبات می‌کنیم که این کار شدنی است و هم نشان می‌دهیم که چگونه و با چه الگوریتمی می‌توان کدها را تولید کرد.

پایه‌ی استقرا: $n = 2$ $(k = 1)$ که بدیهی است.

فرض استقرا: برای $n = 2k - 2$ کد بسته وجود دارد.

حکم استقرا: برای $n = 2k$ هم کد بسته داریم.

بنا به فرض استقرا برای $n = 2k - 2$ کد بسته‌ی $\{S_1, S_2, \cdots, S_{2k-2}\}$ را داریم که می‌توان آن را به صورت دور بسته‌ای مانند شکل زیر نشان داد. رابطه‌ی بین دو کد متمایز را که در یک بیت با هم اختلاف دارند را با خط تیره نشان می‌دهیم (شکل ۱۴.۴).

ما یک بیت صفر به همه‌ی کدهای موجود اضافه کنیم؛ باز هم رابطه‌ی فوق برقرار است. کدهای S_{2k-1} و S_{2k} را به این صورت اضافه می‌کنیم: کد S_{2k} همان کد S_1 است، با این اختلاف که بیت صفر اضافه شده به S_1 یک باشد. کد S_{2k-1} همان کد S_{2k-2} باشد با این اختلاف که بیت صفر اضافه شده به S_{2k-2} یک باشد. با این ترتیب، S_1 با S_{2k}، S_{2k-2} با S_{2k-1}، S_{2k-1} با S_{2k} فقط در یک بیت اختلاف دارند. بنابراین کد حاصل برای $n = 2k$ هم بسته است. □

توجه: در اثبات فوق الگوریتمی ارایه دادیم که در آن، برای اضافه کردن هر دو عنصر به مجموعه‌ی عناصر یک بیت به کدهای قبلی اضافه شد. بنابراین با این الگوریتم، برای $n = 2k$ عنصر کد بسته‌ای با $\frac{n}{2}$ بیت ایجاد می‌کنیم که بهینه نیست.

لم ۵. برای $n = 2^k$ عنصر می‌توان کد گری بسته‌ی k بیتی ایجاد نمود.

اثبات. اثبات به کمک استقرا.

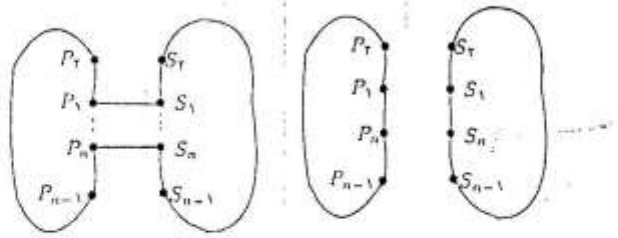
پایه: برای $n = 2$ بدیهی است.

فرض: برای $n = 2^{k-1}$ کد بسته‌ی $k - 1$ بیتی وجود دارد.

حکم: برای $n = 2^k$ کد بسته‌ی k بیتی وجود دارد.

[17] constructive

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۵.۴: ساخت کد گری به طول دو برابر

مجموعه‌ی کدهای بسته‌ی $k-1$ بیتی فرض استقرا (برای $n = 2^{k-1}$ عنصر) را $S = \{S_1, \cdots S_n\}$ می‌نامیم. فرض کنید $P = \{P_1, \cdots P_n\}$ یک کد دیگر مشابه‌ی S است (کدهای P_i و S_i یکسان هستند.) این دو کد بسته را به صورت دور می‌توان نشان داد (شکل ۱۵.۴).

به دور S یک بیت ۱ و به دور P یک بیت ۰ (صفر) اضافه می‌کنیم تا کدهای $S' = \{S'_1, \cdots S'_n\}$ و $S' = \{P'_1, \cdots P'_n\}$ ایجاد شوند. بدیهی‌است که مجموعه‌ی $\{S'_1, \cdots S'_n, P'_n, P'_{n-1}, \cdots P'_1\}$ یک کد بسته و بهینه‌ی k بیتی برای $n = 2^k$ عنصر است. □

لم ۶. برای $n = 2k + 1$ عنصر، کد گری بسته وجود ندارد.

اثبات. فرض کنید که برای این تعداد عنصر یک کد گری بسته داشته باشیم. در این صورت، اگر از یک کد شروع کنیم در یک دور زدن باید به همان کد برسیم. با توجه به آن که از یک کد به کد بعدی فقط یک بیت تغییر می‌کند، باید تعداد تغییرات این بیت‌ها زوج باشد تا بتوان به همان کد اولیه برسیم، یعنی اگر یک بیت صفر به یک تبدیل شود باید در جای دیگر همان بیت یک دوباره به‌صفر تبدیل شود. ولی این کار با توجه به تعداد فرد عناصر ممکن نیست. □

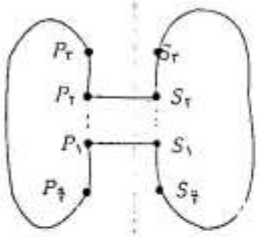
قضیه ۸. برای n عدد می‌توان کد گری $\lceil \log_2 n \rceil$ بیتی ایجاد کرد. اگر n زوج باشد این کد بسته و اگر n فرد باشد باز است.

اثبات. با استقرا اثبات می‌کنیم.

حالت اول: فرض می‌کنیم تعداد عناصر زوج است ($n = 2k$.)

$k = \frac{n}{2}$ زوج است یا فرد. طبق فرض استقرا، اگر k زوج باشد: برای k عنصر کد گری بسته، و اگر فرد باشد، کد گری باز، هر دو به طول $\lceil \log_2 k \rceil$ بیت وجود دارد. اگر k زوج باشد، درست مانند اثبات لم ۵ می‌توان برای n عنصر کد گری بسته‌ی $\lceil \log_2 \frac{n}{2} \rceil + 1 = \lceil \log_2 n \rceil$ بیتی ایجاد کرد. اگر k فرد باشد، دو کد باز و مشابه‌ی $S = \{S_1, \cdots S_k\}$ و $P = \{P_1, \cdots P_k\}$ را برای k عنصر در نظر بگیرید، و فرض کنید که تنها کدهای S_1 و S_2 (و نیز P_1 و P_2) با هم در بیش از یک بیت اختلاف دارند. این دو کد را می‌توان به صورت دو دور «باز» مانند شکل ۱۶.۴ نشان داد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۶.۴: ساخت کد گری

به دور S یک بیت ۱ و به دور P یک بیت ۰ (صفر) اضافه می‌کنیم تا کدهای $S' = \{S'_1, \cdots S'_n\}$ و $S' = \{P'_1, \cdots P'_n\}$ ایجاد شوند. بدیهی‌است که مجموعه‌ی $\{S'_1, \cdots S'_n, P'_n, P'_{n-1}, \cdots P'_1\}$ یک کد بسته است و تعداد بیت‌های آن برابر است با

$$\lceil \log_2 \frac{n}{2} \rceil + 1 = \lceil \log_2 n \rceil$$

حالت دوم: فرض می‌کنیم تعداد عناصر فرد است ($n = 2k + 1$.)

برای $(2k + 2)$ عنصر یک کد بسته‌ی $\lceil \log_2 2k + 2 \rceil$ بیتی ایجاد می‌کنیم. کافی است یک عنصر دلخواه را حذف کنیم؛ کد بسته به کد باز تبدیل و با طول بهینه تبدیل می‌شود، چون $\lceil \log_2 2k + 1 \rceil = \lceil \log_2 2k + 2 \rceil$

□

۲.۲.۴ رابطه‌ی مستقل از حلقه برای اثبات درستی الگوریتم

استقرا، یک راه خوب برای اثبات الگوریتم‌هاست. برای مثال اگر یک الگوریتم حاوی یک حلقه باشد، می‌توانیم بااستقرا روی متغیر حلقه، درستی نتیجه رااثبات کنیم. دراین صورت رابطه‌ی استقرایی برپایه‌ی متغیرهای حلقه به گونه‌ی مستقلی از حلقه (Loop Invariant) بیان می‌گردد. این رابطه را اگر برای همه‌ی مقادیر حلقه اثبات کنیم، در واقع درستی آن را اثبات کرده‌ایم.

جهت روشن‌تر شدن مطلب به مثال زیر توجه کنید:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Proscedure Convert_to_Binary(n);
        Input   : n (A positive number);
        Outpurt : b (Array of bits);
Begin
        t := n;
        k := 0;
        While t > 0 do Begin
                k := k+1;
                b[k] := t mod 2;
                t := t div 2;
        End;
End;
```

الگوریتم فوق عمل تبدیل عدد n را به رشته‌ی دودویی معادل را بر عهده دارد.

برای اثبات درستی این الگوریتم، بر روی k، تعداد دفعاتی که حلقه تکرارشده است، استقرا انجام می‌دهیم. اما قبل از آن باید رابطه‌ای مستقل از حلقه بیابیم که وابسته به متغیرهای حلقه (از جمله k) باشد و استقرا را روی آن برای k های متفاوت انجام دهیم. به رابطه‌ی ذکر شده invariant می‌گوییم.

در این الگوریتم رابطه‌ی استقرایی را به صورت زیر حدس می‌زنیم:

$$n = t2^k + m$$

که در آن m عددی است که آرایه‌ی b نمایش می‌دهد.

اثبات.

پایه‌ی استقرا : اگر $k = 0$ آن‌گاه $m = 0$ پس $n = t2^0 + 0$

فرض استقرا : اگر $k = i$ آن‌گاه $n = t2^i + m$

حکم استقرا: اگر $k = i + 1$ آن‌گاه $n = t'2^{i+1} + m'$ که t' و m' با رابطه‌ی زیر بیان می گردند:

$$t' = \lfloor \frac{t}{2} \rfloor \qquad m' = m + (t \bmod 2)2^i$$

با توجه به روابط فوق و فرض استقرا داریم:

$$n = \lfloor \frac{t}{2} \rfloor 2^{i+1} + m + (t \bmod 2)2^i = 2^i(2\lfloor \frac{t}{2} \rfloor + (t \bmod 2)) + m$$

از طرفی

$$2\lfloor \frac{t}{2} \rfloor + (t \bmod 2) = t$$

پس داریم:

$$n = t2^i + m$$

و این رابطه بنا به فرض استقرا صحیح است.

□

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۳.۲.۴ محاسبه‌ی مقدار یک چندجمله‌ای

هدف محاسبه‌ی مقدار یک چندجمله‌ای $P_n(x)$ از درجه‌ی n بر حسب x است.

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

ورودی: n, x و $a_0, \ldots, a_n$

خروجی: مقدار $P_n(x)$

راه حل اول : چندجمله‌ای را به صورت بازگشتی تعریف می‌کنیم: $P_n(x) = P_{n-1}(x) + a_n x^n$ و مطابق آن الگوریتمی بازگشتی برای حل مسئله می‌نویسیم. در این صورت، هزینه‌ی الگوریتم برابر $n - 1$ عمل جمع خواهد بود به اضافه‌ی تعداد عملیات‌های ضرب که برابر است با:

$$T(n) = T(n-1) + n = \frac{n(n+1)}{2}$$

راه حل دوم : (الگوریتم هورنر[۲۸])
تعریف را به این صورت تغییر می‌دهیم:

$$P'_{n-1}(x) = a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_2 x + a_1$$

$$P_n(x) = xP'_{n-1}(x) + a_0$$

بنابراین:

$$T(n) = T(n-1) + 1 = n$$

```
begin
    p=0;
    for i:=n downto 0 do
        p:=p*x+a[i];
end;
```

۴.۲.۴ زیرگراف القایی بیشینه

هدف به دست آوردن بزرگترین زیرگرافی است که درجه‌ی هر رأس آن حداقل k باشد. به این زیرگراف «زیرگراف القایی بیشینه[۲۹]» می‌گوییم.

[۲۸] Horner's algorithm
[۲۹] Maximal Induced Subgraph

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

یک مثال کاربردی برای این مسئله: عده‌ای را می‌خواهیم به یک مهمانی دعوت کنیم. هریک از این افراد تعدادی از دیگران را می‌شناسد و رابطه‌ی «شناسایی» رابطه‌ای دوطرفه است. می‌خواهیم از بین این افراد، می‌خواهیم بیشترین تعدادی رادعوت کنیم که هرکدامشان لااقل k نفر دیگر را بشناسد. مسئله به صورت گراف مدل می‌شود و هدف به دست آوردن زیرگراف القایی است با درجه‌ی حداقل k.

به عبارت دیگر، در گراف $G = (V, E)$ زیرمجموعه‌ی $S \subset V$ را با حداکثر تعداد رأس‌هامی‌خواهیم پیدا کنیم که زیر گراف القایی آن دارای رأس‌هایی با درجه‌ی حداقل k باشد.

روش پیاده‌سازی: استفاده از ماتریس مجاورت

حل: اگر درجه‌ی همه‌ی گره‌ها از k بیشتر باشد مسئله حل شده است، در غیر این صورت گره‌ای مانند v وجود دارد به طوری که $\text{Degree}(v) < k$. گره‌ی مورد نظر و یال‌های مربوط به آن را حذف می کنیم. این کار را ادامه می‌دهیم تا وقتی که درجه‌ی هر گره‌ی باقی‌مانده حداقل k باشد.

۵.۲.۴ نگاشت یک‌به‌یک

مجموعه‌ی A و تابع $f : A \to A$ داده‌شده است. می‌خواهیم بزرگترین زیرمجموعه‌ی $S \subset A$ را پیدا کنیم که f بر روی آن زیرمجموعه یک‌به‌یک باشد، یعنی $f : S \to S$.

تابع f بر روی A را می‌توان به صورت یک گراف سودار نشان داد. شکل زیر یک تابع را بر روی $A = \{۱, ۲, ۳, ۴, ۵, ۶, ۷\}$ نشان می‌دهد.

هدف پیدا کردن بزرگ‌ترین زیرگرافی است که درجه‌ی ورودی و خروجی هر گره در آن ۱ باشد. این مسئله را می‌توان با استقرا حل کرد.

اگر درجه‌ی ورودی گره‌های این گراف را به دست آوریم، باید گره‌ای به نام $v \in A$ با درجه‌ی ورودی صفر وجود داشته باشد، در غیر این صورت $S = A$ جواب است (چرا؟) v نمی‌تواند عضوی از جواب باشد، پس v و یال‌هایی که از آن خارج می‌شوند را حذف می‌کنیم و در گراف حاصل مسئله را به صورت استقرا حل می‌کنیم.

برنامه‌ی زیر این کار را انجام می‌دهد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Input : f(array[1..n] where values are 1..n)
Output: S a subset of A={1..n}

  begin
     S:=A;
     for j:=1 to n C[j]:=0;  {C[j] is the indegree of j}
     for j:=1 to n Inc(C[F[j]]);
     for j:=1 to n if C[j]=0 then  put j in a queue
     while queue is not empty do
         begin
             remove i from queue;
             S:=S-{i};
             Dec(C[F[i]]);
             if C[F[i]]=0 then  put F[i] in queue;
         end;
  end;
```

۶.۲.۴ مسئله‌ی «ستاره‌ی مشهور»

می‌خواهیم با حداقل تعداد پرسش از نفر n، فردی که ویژگی‌های یک «ستاره‌ی مشهور[۴۰]» را دارد، در صورت وجود، پیدا کنیم. یک نفر ستاره است اگر بقیه او را از قبل بشناسند و او با آن‌ها آشنا نباشد. ما مجاز هستیم از a بپرسیم که آیا b را می‌شناسد؟ این یک پرسش است. توجه کنید در این مسئله رابطه‌ی شناختن یک رابطه‌ی یک‌طرفه است.

چون $n(n-1)/2$ گروه دو نفری داریم پس اگر پرسش‌ها به طور دل‌خواه باشند، در بدترین حالت نیاز به $n(n-1)$ پرسش داریم.

راه حل اول: فرض کنیم که ما می‌توانیم ستاره را بین $n-1$ نفر اول توسط استقرا به دست آوریم. چون حداکثر یک ستاره می‌توانیم داشته باشیم، سه حالت ممکن است اتفاق بیفتد:

۱. ستاره بین $n-1$ نفر اول است.

۲. نفر nام ستاره است.

۳. ستاره نداریم.

در حالت اول فقط باید بررسی کنیم که نفر nام ستاره را می‌شناسد و ستاره او را نمی‌شناسد. در دو حالت بعد، حداکثر به $2(n-1)$ پرسش نیاز داریم؛ چرا که باید روشن کنیم که آیا هریک از $n-1$ نفر بقیه نفر nام می‌شناسد و نفر nام او را نمی‌شناسد. بنابراین، جمع کل پرسش‌ها $n(n-1)$ است که در بدترین حالت هم به آن رسیدیم.

راه‌حل بهتر: به روش حذفی عمل می‌کنیم، در هر پرسش یک نفر را از مجموعه حذف کنیم و با استفاده از استقرا راه حل را دنبال می‌کنیم. برای این کار فرض کنید که ما از A بپرسیم که آیا B را می‌شناسد یا نه؟ اگر جواب مثبت باشد، A نمی‌تواند ستاره باشد، و اگر جواب منفی باشد B نمی‌تواند ستاره باشد؛ در هر صورت یکی از این افراد

[۴۰] celebrity

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

حذف می‌شوند. حال کافی است بین $n-1$ نفر باقی‌مانده ستاره را پیدا کنیم. با $n-1$ پرسش به یک نفر به نام s می‌رسیم؛ تنها s می‌تواند ستاره باشد، و با $2(n-1)$ پرسش این امر را می‌توان مشخص کرد. بنابراین تعداد کل پرسش‌ها می‌شود:

$$n-1+2(n-1)=3(n-1)$$

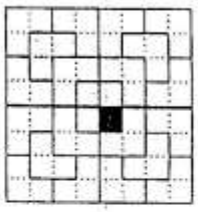
البته این تعداد پرسش‌ها کمینه نیست و می‌توان کمی به‌تر از این عمل کرد.

حال الگوریتم این راه حل را می‌نویسیم. ورودی Know یک ماتریس مجاورت $n \times n$ است. مقدار Know[i,j] برابر یک است اگر فرد i فرد j را بشناسد، وگرنه صفر است. هدف پیدا کردن شماره‌ی s است به گونه‌ای که تمامی عناصر ستون کام (به جز[s,s]) یک و تمامی عناصر سطر کام (به جزمولفه [s,s]) صفر باشد:

```
Algorithm celebrity (Know);
Input:Know (an n*n Boolean matrix).
output:celebrity.

  begin
     i:=1;
     j:=2;
     next:=3;
     {in the first phase we eliminate all but one candidate}
     while  next <= n+1 do
        if Know[i,j] then  i:=next
        else j:=next;
        next:=next+1;
        {one of either i or j is eliminated}
     if i=n+1 then  candidate:=j;
              else  candidate:=i;
     {Now we check that the candidate is indeed the celebrity}
     wrong:=false;
     k:=1;
     Know[candidate,candidate]:=false;
         {a dummy variable to pass the test}
     while not wrong and k<=n do
        if Know[candidate,k] then wrong:=true;
        if not Know[k,candidate] then
            if candidate<>k then wrong:=true;
        k:=k+1;
     if not wrong then celebrity:=candidate
     else celebrity:=0 {no celebrity}
  end;
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۱۷.۴: فرش‌کردن صفحه‌ی شطرنجی.

۳.۴ روش تقسیم‌وحل برای طراحی الگوریتم‌ها

این روش که معمولاً برای به دست آوردن الگوریتم‌های سریع در حل مسئله‌ها مورد استفاده قرار می‌گیرد، دارای دو مرحله‌ی «تقسیم» و «حل» است. در مرحله‌ی اول، مسئله به چند زیرمسئله‌ی کوچک‌تر تقسیم می‌شود. این زیرمسئله‌ها به صورت بازگشتی حل می‌شوند و از ترکیب حل آن‌ها جواب مسئله به دست می‌آید. واژه‌ی انگلیسی این روش divide and conquer است که ترجمه‌ی «تفرقه بینداز و حکومت کن» برای آن بسیار آشناست.

این روش مشابه‌ی روش طراحی الگوریتم با استفاده از استقراست. البته معمولاً در روش استقرایی مسئله به یک مسئله با اندازه‌ی کوچک‌تر تقسیم می‌شود.

توضیحات بیشتر را در قالب مثال‌هایی ارایه می‌کنیم.

۱.۳.۴ فرش‌کردن صفحه‌ی شطرنجی

صفحه‌ی شطرنجی به ابعاد $2^k \times 2^k$ و موزاییکی به شکل L داریم. می‌خواهیم این صفحه را طوری با این موزاییک فرش کنیم. چون برای هر k داریم $2^{2k} = 3Al + 1$، این صفحه را اگر فرش کنیم در به‌ترین حالت یکی از خانه‌ها خالی می‌ماند. می‌خواهیم نشان دهیم که می‌توان باانتخاب هر خانه‌ی دل‌خواه به عنوان خانه‌ی خالی، بقیه‌ی صفحه را می‌توان کاملاً فرش کرد، به گونه‌ای که هیچ موزاییکی از جدول بیرون نزند. البته در این فرش‌کردن ما مجاز به دوران موزاییک‌ها هستیم.

راه‌حل. بر روی k استقرا انجام می‌دهیم. برای $k = 1$ بدیهی است با چرخاندن موزاییک به جواب می‌رسیم. مربع $2^k \times 2^k$ را مطابق شکل ۱۷.۴ به چهار قسمت تقسیم می‌کنیم. طبق استقرا هر مربع را می‌توانیم طوری فرش کنیم که هر خانه‌ی دل‌خواه آن خالی بماند. خانه‌ی خالی خواسته شده در مسئله درون یکی از این چهار مربع است. این مربع به اندازه‌ی $2^{k-1} \times 2^{k-1}$ را با استقرا طوری فرش می‌کنیم تا خانه‌ی مورد نظر خالی بماند. حال، سه مربع دیگر را طوری فرش می‌کنیم تا خانه‌ی گوشه‌ی محل تلاقی آن‌ها خالی بماند، سه خانه‌ی خالی باقی مانده را نیز توسط یک موزاییک فرش می‌کنیم و مساله حل است.

۲.۳.۴ زمان‌بندی دوره‌ی بازی‌ها

برای n تیم می‌خواهیم یک دوره بازی با حداقل مدت طراحی کنیم که در آن هر تیم با بقیه‌ی تیم‌ها بازی کند، با این شرط که هر تیم در هر روز بیش از یک بازی انجام ندهد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

باید به پرسش‌های زیر پاسخ دهیم:

الف) حداقل در چند روز بازی‌ها قابل انجام است؟

ب) برنامه‌ی زمان‌بندی بازی‌ها چیست؟

تعداد کل بازی‌ها برابر $n(n-1)/2$ است و در هر روز حداکثر $\lfloor n/2 \rfloor$ تعداد بازی می‌تواند انجام شود. پس حد پایین تعداد روزهای دوره‌ی بازی اگر n زوج باشد برابر $n-1$ و اگر n فرد باشد برابر n روز است.

یک برنامه‌ی زمان‌بندی بازی‌ها برای n تیم در m روز را می‌توان با یک ماتریس A به ابعاد $n \times m$ مدل کرد. درایه‌ی $A[i,j]=k$ یعنی تیم i با k در روز j بازی می‌کند. در این ماتریس هم‌چنین،

۱. در سطر i تمام اعداد ۱ تا n به جز i قرار دارند.

۲. درهیچ سطر و ستونی عدد تکراری نداریم.

۳. رابطه‌ی زیر صادق است:

$$A[b,i] = a \Leftrightarrow A[a,i] = b$$

ابتدا مسئله را برای $n = 2^k$ حل می‌کنیم. ادعا می‌کنیم می‌توان بازی‌ها را در $n-1$ روز انجام داد. این ادعا را با استقرا اثبات می‌کنیم.

برای دو تیم واضح است که در یک روز بازی انجام می‌شود. n تیم را به دو گروه $n/2$تایی تقسیم می‌کنیم. طبق فرض استقرا، هر گروه می‌تواند بازی‌های بین خود را در $n/2-1$ روز انجام دهد. حال فقط بازی‌های هر تیم از گروه دوم با تیم‌های گروه اول (و بر عکس) باقی مانده است. از این بازی‌ها می‌توان برنامه‌ی بازی‌های هر تیم از گروه اول را هم به دست آورد. مدل ماتریسی بازی‌های باقی‌مانده مطابق شکل ۱۸.۴ است. اگر بخواهیم کل بازی‌ها را در $n-1$ روز انجام دهیم، این ماتریس باید یک ماتریس مربعی به ابعاد $n/2 \times n/2$ باشد به طوری که در آن تنها اعداد ۱ تا $n/2$ آمده باشد. هم‌چنین در هر سطر و در در هر ستون این ماتریس عدد تکراری نباید داشته باشیم. این چنین ماتریسی به «مربع لاتین[۳۱]» مشهور است و برای تولید آن الگوریتم‌های متعددی وجود دارد. یکی از این راه‌حل‌ها این است: اعداد ۱ تا $n/2$ را در سطر اول می‌نویسیم و اعداد هر سطر بعدی را از با دوران به راست (یا چپ) سطر قبلی‌اش به اندازه‌ی یک خانه به دست می‌آوریم. به مثال شکل ۱۸.۴.ب مراجعه کنید. از این مربع لاتین، مربع لاتین بالای آن (برنامه‌ی بازی‌های تیم‌های گروه اول با دوم هم به دست می‌آید.

به این صورت، کل بازی‌ها در $n/2 - 1 + n/2 = n - 1$ روز انجام می‌شود.

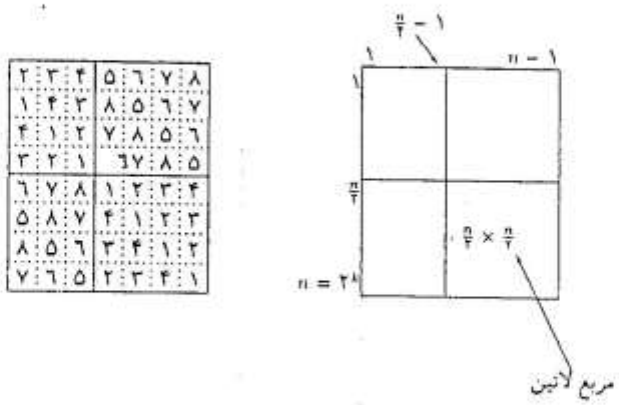
حال اگر n توانی از دو نباشد. اگر n فرد بود، با اضافه کردن یک تیم «اضافی» حل مسئله مانند حالت زوج می‌شود و یک روز به زمان بازی‌ها اضافه می‌گردد. در زمان‌بندی‌ای که به دست می‌آید، هر تیم که با تیم «اضافی» بازی دارد، در واقع استراحت می‌کند.

برای حل در حالت کلی هم از استقرا استفاده می‌کنیم. فرض می‌کنیم حل مسئله را برای $n/2$ تیم داریم. دو حالت وجود دارد:

الف) $n/2$ زوج است. در این صورت دقیقاً مانند حالت $n = 2^k$ عمل می‌کنیم. هر گروه بازی‌های خودش را در $n/2-1$ روز انجام می‌دهد و برنامه‌ی بازی‌های بین دو گروه از طریق مربع لاتین به دست می‌آید.

[۳۱] latin square

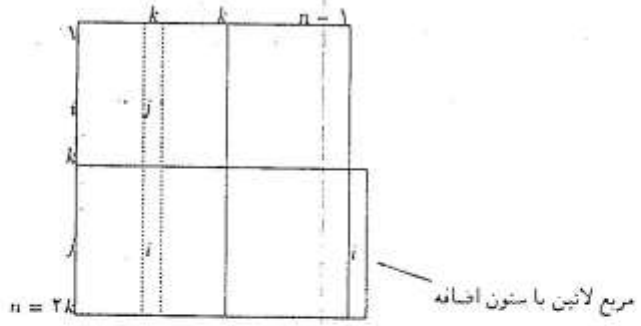
seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



۲	۳	۴	۵	۶	۷	۸	
۱	۴	۳	۸	۵	۶	۷	
۴	۱	۲	۷	۸	۵	۶	
۳	۲	۱	۶	۷	۸	۵	
۶	۷	۸	۱	۲	۳	۴	
۵	۸	۷	۴	۱	۲	۳	
۸	۵	۶	۳	۴	۱	۲	
۷	۶	۵	۲	۳	۴	۱	

(الف) (ب) مثال برای $n = 4$

شکل ۱۸.۴: زمان‌بندی دوره‌ی بازی‌ها برای $n = 2^k$ تیم.



شکل ۱۹.۴: حالتی که $n/2$ فرد است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

	۱	۲	۳
۱	۲	۳	۴
۲	۱	۴	۳
۳	۴	۱	۲
۴	۳	۲	۱

۱
۲

شکل ۲۰.۴: برنامه‌ی بازی‌ها برای $n = ۲$ و $n = ۴$.

	۱	۲	۳	۴	۵	
۱	۲	۳	۴	۶	۵	
۲	۱	۵	۳	۴	۶	
۳	۶	۱	۲	۵	۴	
۴	۵	۶	۱	۲	۳	۱
۵	۴	۲	۶	۳	۱	۲
۶	۳	۴	۵	۱	۲	۳

	۱	۲	۳
۱	۲	۳	-
۲	۱	-	۳
۳	-	۱	۲

شکل ۲۱.۴: برنامه‌ی بازی‌ها برای $n = ۳$ و $n = ۶$.

ب) $n/۲$ فرد است. در این صورت هر گروه بازی‌های بین خودشان رادر $n/۲$ روز انجام می‌دهند. با توجه به این که در هر روز از این $n/۲$ روز (طبق استقرا) یک تیم از هر گروه قرار است استراحت کند، به جای استراحت این تیم‌ها(یکی از گروه اول و دیگری از گروه دوم) می‌توانند با هم بازی کنند.

با این ترتیب، تعداد روزهای باقی‌مانده برای بازی تیم‌های گروه اول با گروه دوم $n/۲ - ۱$ است. مجدداً ماتریس C به ابعاد $n/۲ \times (n/۲ - ۱)$ برای بازی‌های تیم‌های گروه دوم با گروه اول را در نظر بگیرید (شکل ۱۹.۴.) این ماتریس باید حاوی اعداد ۱ تا $n/۲$ باشد با این شرط که در هر سطر و ستون آن اعداد تکراری نداشته باشیم. هم‌چنین برای هر تیم i از گروه دوم، می‌دانیم که این تیم در $n/۲$ روز اول بازی‌ها، دقیقاً با یک تیم j از گروه اول بازی کرده‌است. این به معنی آن است که در سطر iام ماتریس C باید همه‌ی اعداد ۱ تا $n/۲$ بیاید به جز j.

برای اعمال آخرین شرط فوق برای ماتریس C، یک ستون به آن اضافه می‌کنیم تا مربع شود و در سطر iام این ستون عدد j می‌نویسیم (i و j در بالا تعریف شده‌اند.) حال ماتریس C با ستون اضافی به یک مربع لاتین تبدیل می‌شود که یکی از ستون‌های آن از قبل مشخص است. اگر این مربع را ۹۰ درجه بچرخانیم می‌توانیم همان الگوریتم تولید مربع لاتین، که در بالا گفته شد، با این تفاوت که در سطر اول جای‌گشتی از اعداد ۱ تا

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

	۱	۲	۳	۴	۵	۶	۷	۸	۹	
۱	۲	۳	۴	6	۵	۱۰	۹	۸	۷	
۲	۱	۵	۳	۴	7	۶	۱۰	۹	۸	
۳	8	۱	۲	۵	۴	۷	۶	۱۰	۹	
۴	۵	9	۱	۲	۳	۸	۷	۶	۱۰	
۵	۴	۲	10	۳	۱	۹	۸	۷	۶	
۶	۷	۸	۹	1	۱۰	۲	۳	۴	۵	1
۷	۶	۱۰	۸	۹	2	۳	۴	۵	۱	2
۸	3	۶	۷	۱۰	۹	۴	۵	۱	۲	3
۹	۱۰	4	۶	۷	۸	۵	۱	۲	۳	4
۱۰	۹	۷	5	۸	۶	۱	۲	۳	۴	5

۲	۳	۴	—	۵
۱	۵	۳	۴	—
—	۱	۲	۵	۴
۵	—	۱	۲	۳
۴	۲	—	۳	۱

شکل ۲۲.۴: برنامه‌ی بازی‌ها برای $n = ۵$ و $n = ۱۰$.

$n/۲$ قرار دارد، را اعمال کنیم و ماتریس C را به دست آوریم. مانند قبل، قسمت چهارم ماتریس کلی را می‌توانیم از C تولید کنیم.

در زیر مثال برای $n = ۱۰$ آورده شده است. ابتدا باید مسئله را برای ۵ تیم حل کنیم و برای آن باید ماتریس را برای $n_0 = ۶$ به دست آوریم.

$$۱۰/۲ = ۵ \Rightarrow ۵ + ۱ = ۶ \Rightarrow ۶/۲ = ۳ \Rightarrow ۳ + ۱ = ۴ \Rightarrow ۴/۲ = ۲$$

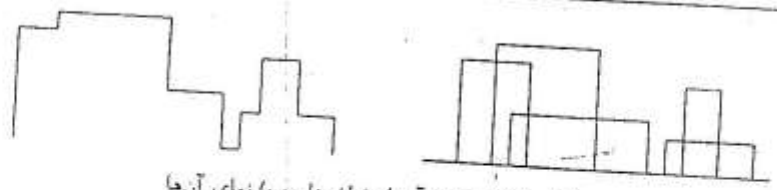
پس ابتدا برای $n_0 = ۲$ و سپس برای $n = ۴$ مسئله را حل می‌کنیم (شکل ۲۰.۴.)
آن‌گاه برای $n_0 = ۳$ و سپس برای $n = ۶$ مسئله را حل می‌کنیم (شکل ۲۱.۴.)
و در نهایت ، مسئله را برای $n = ۵$ و سپس برای $n = ۱۰$ حل می‌کنیم (شکل ۲۲.۴.)

۳.۳.۴ مسئله‌ی برج‌ها

n برج داده شده‌اند. همه‌ی برج‌ها بر روی محور xها قرار دارند و برج i با سه عدد a_i, b_i, c_i مشخص می‌شود، که در آن a_i و b_i به ترتیب مختصات نقطه‌ی چپ و پایین، راست و پایین و c_i ارتفاع برج نام است. می‌خواهیم «نمای برج‌ها»، آن چه از دور دیده می‌شود را به دست بیاوریم. نمای برج‌ها، لبه‌ی خارجی حاصل از کنار هم قرار گرفتن برج‌هاست که دیده می‌شود.

یک نما را می‌توان به صورت دنباله‌ی $< x_1, h_1, x_2, h_2, \ldots, x_{k-1}, h_{k-1}, x_k >$ نمایش داد که در آن x_iها مختصات گوشه‌های تعدادی از برج‌هاست (aها یا bها) که دیده‌می‌شوند. این xها مرتب می‌باشند ($x_1 < x_2 < \cdots < x_k$.) هم‌چنین ارتفاع نما بین x_i و x_{i+1} برابر h_i است و ارتفاع نما برای $x < x_1$ و $x > x_k$ صفر است. برای مثال شکل ؟؟ نمای چند برج را نشان می‌دهد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲۳.۴: الف) آسمان‌خراش‌ها و ب) نمای آن‌ها

راه‌حل ۱– برای $n = 1$ دنباله همان سه عدد تنها برج است.
برای n برج، مسئله را برای $n-1$ به صورت استقرا حل می‌کنیم و $S = < x_1, h_1, x_2, h_2, \ldots, x_{k-1}, h_{k-1}, x_k >$ دنباله‌ی آن را به دست می‌آوریم. حال ساختمان nام را با مشخصات a_n, b_n, c_n به این نما اضافه می‌کنیم.
برای این کار کافی‌است x_i و x_jای را در نما پیدا کنیم که $x_i \le a_n < x_{i+1} \le x_{j-1} < b_n < x_j$ (حالت تساوی، حالت خاص است.) حال اگر $h_i < c_n$، زوج $< a_n, c_n >$ را پس از $< x_i, h_i >$ در S درج می‌کنیم. هم‌چنین اگر $h_{j-1} < c_n$، زوج $< b_n, c_n >$ را پس از $< x_{j-1}, h_{j-1} >$ در S درج می‌کنیم. حال ارتفاع‌های h_{i+1}, h_{i+2}، تا h_{j-1} را به ترتیب با c_n مقایسه می‌کنیم و اگر برای $i < r < j$، $h_r < c_n$ بود، در S، h_r را برابر c_n قرار می‌دهیم. تنها مرحله‌ای که باقی‌مانده است حذف نقاط متوالی هم‌ارتفاع در S است، که با به راحتی قابل انجام است (البته این کار را در همان پویش اول هم می‌توان انجام داد.)
برای ساده‌تر شدن پیاده‌سازی فوق به‌تر است به اول دنباله‌ی S زوج $< -\infty, 0 >$ و به آخر دنباله زوج $< +\infty, 0 >$ را اضافه کنیم.

پیچیدگی الگوریتم

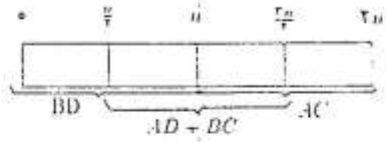
پیداکردن a_n و b_n در دنباله‌ی S، با $\Theta(\log k)$ امکان‌پذیر است. ولی تصحیح ارتفاع‌ها می‌تواند حداکثر $\Theta(k)$ باشد.
چون $k \le n$ داریم:

$$T(n) = T(n-1) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$$

راه‌حل ۲– برای این مسئله از طریق تقسیم‌وحل راه حل کارائری وجود دارد. ابتدا مسئله‌ی باندازه‌ی n را به دو مسئله با اندازه‌های $\lceil n/2 \rceil$ $\lfloor n/2 \rfloor$ تقسیم می‌کنیم و هر دو زیرمسئله را به همین روش حل می‌کنیم.
فرض کنید نمای زیر مسئله‌ی اول $S_1 = < x_1, h_1, x_2, h_2, \ldots, x_{k-1}, h_{k-1}, x_k >$ و نمای زیر مسئله‌ی دوم $S_2 = < x_r, h_r, x_{r+1}, h_{r+1}, \ldots, x_{p-1}, h_{p-1}, x_p >$ باشد.
برای به دست آوردن نمای مسئله باید S_1 و S_2 را در هم ادغام کرد. این کار درست مانند عمل ادغام دو آرایه‌ی مرتب در MergeSort است: بین دو دنباله، کوچکترین x را انتخاب می‌کنیم و آن را همراه با ارتفاع بعدیش در دنباله جواب می‌نویسیم. حال اعمال زیر را تا پایان یافتن دنباله‌ها تکرار می‌کنیم: سراغ x بعدی می‌رویم (بین دو x کنونی دو دنباله، x کوچک‌تر را انتخاب می‌کنیم.) و اگر ارتفاع آن بیش‌تر از آخرین ارتفاع نوشته شده در دنباله‌ی جدید است، آن x را به همراه بعدیش در دنباله‌ی جواب می‌نویسیم.
پیچیدگی الگوریتم: قسمت مهم این الگوریتم ادغام دو دنباله‌است که مانند ادغام دو آرایه‌ی مرتب از مرتبه‌ی $\Theta(\lceil \frac{n}{2} \rceil + \lfloor \frac{n}{2} \rfloor = n)$ است. پس:

$$T(n) = 2T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲۴.۴: مدل‌سازی مسئله‌ی ضرب دو چندجمله‌ای.

که سریع‌تر از قبلی می‌باشد.

می‌توانیم استدلال کنیم که حد پایین این مسئله $n \log n$ است. اگر دنباله ورودی شامل برج‌هایی باشد که هیچ دوتایی از آن‌ها باهم تداخل نداشته باشند باحل کردن مسئله‌ی برج‌ها دنباله‌ای مرتب شده از x_iها به دست می‌آید. اگر این عمل را در زمانی به‌تر از $n \log n$ انجام دهیم، به الگوریتم مرتب‌کننده‌ی مبتنی بر مقایسه‌ای رسیده‌ایم که در زمانی به‌تر از $n \log n$ مرتب می‌کند. این غیر ممکن است!

۴.۳.۴ ضرب دو چندجمله‌ای

دو چندجمله‌ای P و Q برحسب x و از درجه‌ی n داده شده‌اند:

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

$$Q_n(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_0$$

می‌خواهیم با یک الگوریتم کارا حاصل‌ضرب $R_{2n}(x) = P_n(x)Q_n(x) = \sum_{i=0}^{2n} c_i x^i$ را محاسبه کنیم. پس در این مسئله،

ورودی: $n, a_n, a_{n-1}, \ldots, a_0, b_n, b_{n-1}, \ldots, b_0$

خروجی: $c_{2n}, c_{2n-1}, \ldots, c_0$

برای پیاده‌سازی می‌توانیم ورودی را دو آرایه‌ی $n + 1$ عنصری A و B را و خروجی را در آرایه‌ی $2n + 1$ عنصری C بنویسیم (شکل ۲۴.۴).

راه‌حل ۱ — برای $0 \le k \le 2n$ داریم، $c_k = \sum_{i+j=k} a_i b_j$. به این ترتیب، cها را می‌توان با $\Theta(n^2)$ به دست آورد.

```
for i:=0 to 2*n do      C[i]:=0;
for i:=0 to n do
    for j:=0 to n do
        C[i+j]:=C[i+j]+A[i]*B[j];
```

راه‌حل ۲ — ما با استفاده از تقسیم‌وحل، سعی می‌کنیم روشی کاراتر برای حل این مسئله بیابیم. در این جا فرض می‌کنیم $n = 2^k$. هرچند الگوریتم در حالت کلی هم درست است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

در ابتدا P_n و Q_n را به دو نیمه تقسیم می‌کنیم:

$$P_n(x) = A_{\frac{n}{2}} x^{\frac{n}{2}} + B_{\frac{n}{2}}$$

$$Q_n(x) = C_{\frac{n}{2}} x^{\frac{n}{2}} + D_{\frac{n}{2}}$$

که در آن A، B، C و D چندجمله‌ای‌هایی از درجه‌ی $\frac{n}{2}$ هستند. حال، حاصل‌ضرب PQ برابر است با:

$$PQ(x) = A_{\frac{n}{2}} C_{\frac{n}{2}} x^{n} + (A_{\frac{n}{2}} D_{\frac{n}{2}} + B_{\frac{n}{2}} C_{\frac{n}{2}}) x^{\frac{n}{2}} + B_{\frac{n}{2}} D_{\frac{n}{2}}$$

در این روش برای به دست آوردن PQ باید چهار بار و هربار دو چندجمله‌ای از درجه‌ی $\frac{n}{2}$ را به صورت بازگشتی در هم ضرب کنیم و ضرایب حاصل را با توجه به میزان «شیفت nk» که از ضرب در x^k به دست می‌آید با هم جمع نماییم. به عبارت دیگر، $A_{\frac{n}{2}} C_{\frac{n}{2}}$ یعنی به دست آوردن بردار ضرایب چندجمله‌ای حاصل از ضرب A و B. جمع این ضرایب با مقادیر موجود در آرایه‌ی C پس از اعمال شیفت n.

برای تحلیل این الگوریتم داریم:

$$T(n) = 4T(\frac{n}{2}) + \Theta(n)$$

که در نتیجه با استفاده از قضیه‌ی اصلی $T(n) = \Theta(n^2)$ است. پس الگوریتم کارا تر نشد.

راه‌حل ۲– ما می‌توانیم این روش را به‌تر کنیم. به اتحاد زیر توجه کنید:

$$AD + BC = (A - B)(D - C) + AC + BD$$

اگر به جای محاسبه‌ی $AD + BC$ از اتحاد بالا استفاده کنیم، می‌توانیم حاصل‌ضرب‌های BD و AC را ذخیره کنیم و در جای دیگر از آن استفاده کنیم. در نتیجه فقط نیاز به ۳ بار ضرب دوتا چندجمله‌ای از درجه‌ی $\frac{n}{2}$ داریم. در این صورت خواهیم داشت:

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

و با استفاده از قضیه‌ی اصلی داریم: $T(n) = \Theta(n^{\log_2 3})$. البته، تاثیر این کارایی در n های بالا مشهود است.

این تکنیک را آقای استراسون[۳۲] برای ضرب دو ماتریس هم به کار برده است. ضرب دو ماتریس در کارهای مهندسی کاربرد زیادی دارد. در روش عادی از مرتبه‌ی n^3 است. اما با روش استراسون به درجه‌ی $n^{\log_2 7}$ کاهش یافته است. و اخیرا نیز به $n^{2.37}$ رسیده است.

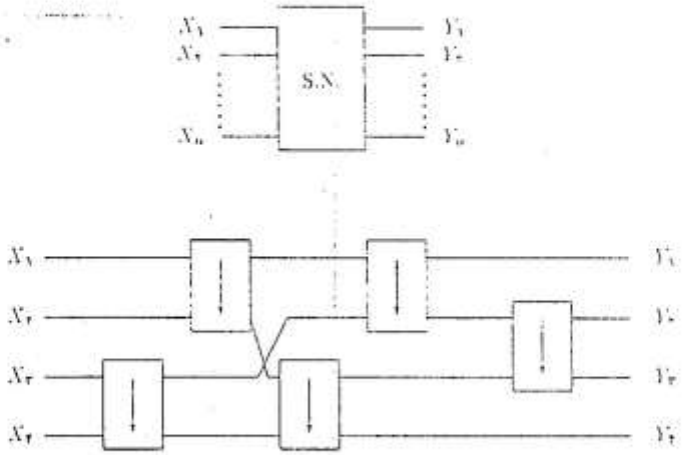
۵.۳.۴ شبکه‌های مرتب‌ساز

می‌خواهیم با استفاده از مقایسه‌کننده‌ها به عنوان المان اصلی، مداری بسازیم تا با حداقل تعداد مقایسه‌کننده‌ها و نیز حتی‌الامکان با حداقل تاخیر n عدد چند بیتی را مرتب نماید.

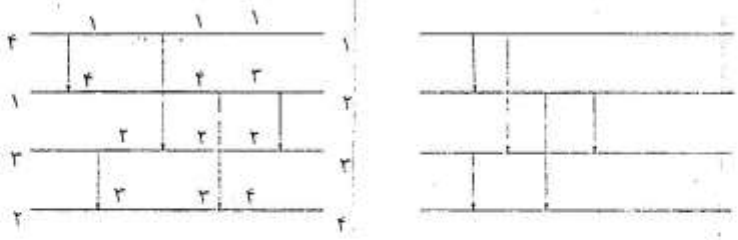
چرا مدار شکل ۲۶.۴ درست کار می‌کند؟ آیا باید همه‌ی $n!$ جایگشت را بررسی کنیم؟ شکل ۲۷.۴ مدار یک مقایسه‌کننده‌ی دوتایی را نشان می‌دهد.

[۳۲] Strassen

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

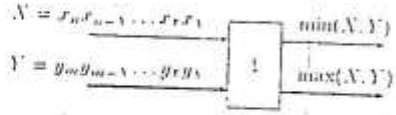


شکل ۲۵.۴: شمای کلی یک شبکه‌ی مرتب‌ساز.



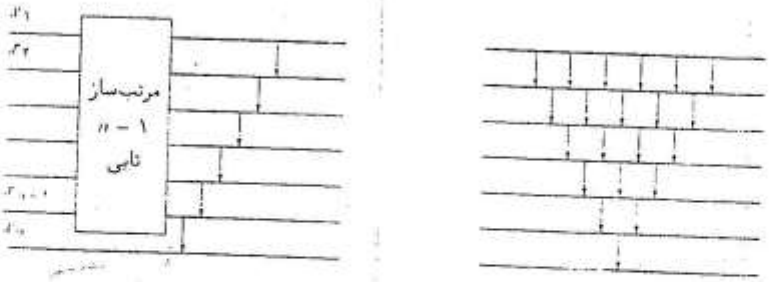
شکل ۲۶.۴: دو شبکه‌ی مرتب‌ساز برای ۴ = n. مقایسه‌کننده‌ها با علامت ↓ نشان داده شده‌اند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



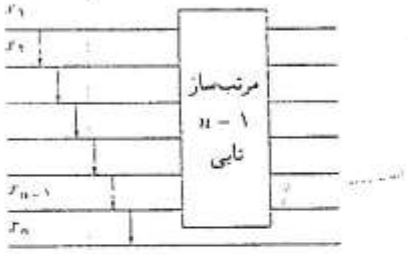
Current State	input	Next State	output
0	00	0	00
0	01	1	01
0	10	2	01
0	11	0	11
1	x y	1	x y
2	x y	2	y x

شکل ۲۷.۴: مدار مقایسه‌کننده‌ی دوتایی.



شکل ۲۸.۴: مدار مقایسه‌کننده‌ی مبتنی بر درج.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۲۹.۴: مدار مقایسه‌کننده‌ی مبتنی بر انتخاب.

روشهای مختلف طراحی شبکه‌های مرتب‌ساز

الف – مبتنی‌بر درج (مانند insertion sort) مانند شکل ۲۸.۴.

تحلیل:

$S(n)$: اندازه‌ی شبکه (تعداد مقایسه‌کننده‌ها)

$T(n)$: زمان تاخیر مدار؛ یعنی حداکثر تعداد مقایسه‌کننده‌ای که یک عدد ورودی باید از آن عبور کند تا به خروجی برسد. برای این مدار داریم:

$$S(n) = \begin{cases} ۰ & n = ۱ \\ S(n-۱) + n - ۱ \end{cases}$$

$$\Longrightarrow S(n) = S(۱) + ۱ + ۲ + \cdots + n - ۱ = \frac{n(n-۱)}{۲}$$

$$T(n) = \begin{cases} ۰ & n = ۱ \\ T(n-۱) + ۲ \end{cases} \Longrightarrow T(n) = ۲n - ۳$$

ب – مبتنی بر انتخاب (مانند bubble sort) مانند شکل ۳۰.۴.

اندازه و زمان این شبکه هم مانند شبکه‌ی قبلی است.

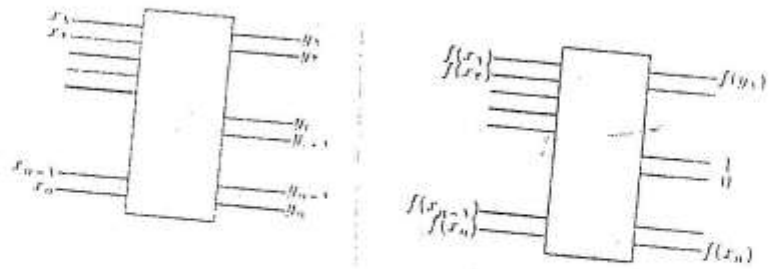
اثبات درستی مدارهای قبل ساده است؛ ولی برای مدارهای دیگر چطور؟

قضیه‌ی صفر و یک (Zero and One Principle)

قضیه ۹. شرط لازم و کافی برای این که یک شبکه‌ی مرتب‌ساز درست کار کند این است که کلیه‌ی ترتیب‌های n صفر و یک را به صورت غیر نزولی مرتب کند.

اثبات. اثبات شرط لازم بدیهی است. برای شرط کافی، فرض کنید که $f(x)$ یک تابع یکنواخت است به طوری که $f(x) \leq f(y)$ اگر $x \leq y$. در آن صورت اگر شبکه‌ای جای‌گشت $(x_۱, \ldots, x_n)$ را به $(y_۱, \ldots, y_n)$ تبدیل کند،

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳۰.۴: اثبات قضیه‌ی صفر و یک.

آن‌گاه $(f(x_1),\ldots,f(x_n))$ را نیز به $(f(y_1),\ldots,f(y_n))$ تبدیل می‌نماید (شکل ؟؟.) حال با استفاده از برهان خلف فرض کنید که برای یک i، $y_i > y_{i+1}$ است. در این صورت تابع f را به صورت زیر انتخاب می‌کنیم:

$$f(x) = \begin{cases} 0 & x < y_i \\ 1 & x \geq y_i \end{cases}$$

□

در این صورت در خروجی ۱ جلوی ۰ خواهد بود که مخالف فرض قضیه است.

ترکیب‌ساز زوج‌فرد (Odd-Even Merger)

شکل ۳۱.۴ نحوه‌ی ساختن این شبکه را نشان می‌دهد. فرض بر این است که $a_1,\ldots,a_n$ و $b_1,\ldots,b_n$ به صورت غیر نزولی مرتب هستند.

قضیه ۱۰. اگر $\bar{a} = a_1,\ldots,a_n$ و $\bar{b} = b_1,\ldots,b_n$ به صورت غیر نزولی مرتب باشند، $\bar{c} = c_1,\ldots,c_n$ در شبکه‌ی شکل ۳۱.۴) به صورت غیر نزولی مرتب خواهد بود.

اثبات. با استفاده از قضیه‌ی صفر و یک، فرض می‌کنیم که $\bar{a}$ و $\bar{b}$ به صورت‌های زیرند:

$$\bar{a} = a_1,\ldots,a_n = \overbrace{00\ldots0}^{x}\overbrace{11\ldots1}^{n-x}$$

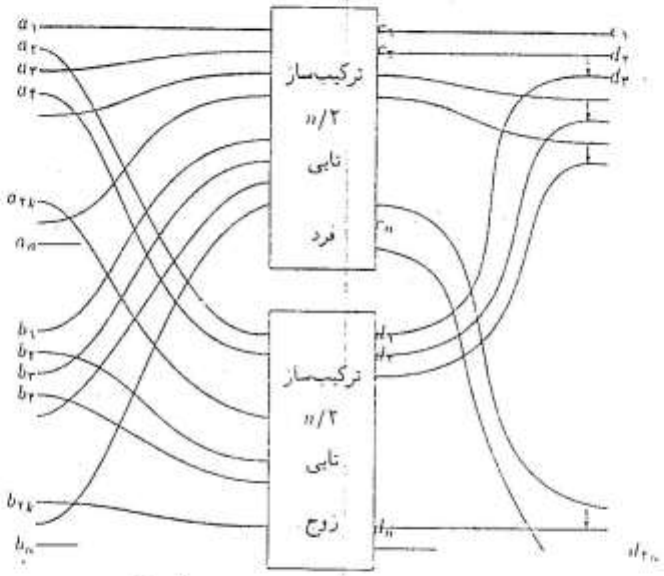
$$\bar{b} = b_1,\ldots,b_n = \overbrace{00\ldots0}^{y}\overbrace{11\ldots1}^{n-y}$$

با استفاده از فرض استقرا، داریم:

$$\bar{c} = \underbrace{\overbrace{00\ldots0}^{x+y-n/2}\,11\ldots11}_{n}$$

seyed hamed hoseinian
Software Engineering pnub

saeeditpnu.blogfa.com



شکل ۳۱.۴: ترکیب‌ساز زوج‌فرد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

$$\bar{d} = \overbrace{\text{۰۰}\cdots\text{۰}}^{\lceil f/2 \rceil + \lfloor g/2 \rfloor}\underbrace{\text{۱۱}\cdots\text{۱۱}}_{n}$$

پس اختلاف تعداد صفرهای $\bar{c}$ و $\bar{d}$ برابر است با: $(\lceil f/2 \rceil + \lfloor g/2 \rfloor) - (\lfloor f/2 \rfloor + \lceil g/2 \rceil)$ و این مقدار برابر ۰، ۱ و یا ۲ است. این سه حالت را بررسی می‌کنیم:

۱. تعداد صفرهای $\bar{c}$ و $\bar{d}$ برابر هستند:

$$\left.\begin{array}{l} \bar{c} = \text{۰۰۰}\cdots\text{۰۰۱۱۱}\cdots\text{۱} \\ \bar{d} = \text{۰۰۰}\cdots\text{۰۰۱۱۱}\cdots\text{۱} \end{array}\right\} \Longrightarrow \bar{c} \text{ مرتب است}$$

۲. اختلاف تعداد صفرهای $\bar{c}$ و $\bar{d}$ برابر ۱ است:

$$\left.\begin{array}{l} \bar{c} = \text{۰۰۰}\cdots\text{۰۰۱۱۱}\cdots\text{۱} \\ \bar{d} = \text{۰۰۰}\cdots\text{۰۱۱۱۱}\cdots\text{۱} \end{array}\right\} \Longrightarrow \bar{c} \text{ مرتب است}$$

۳. اختلاف تعداد صفرهای $\bar{c}$ و $\bar{d}$ برابر ۲ است:

$$\left.\begin{array}{l} \bar{c} = \text{۰۰۰}\cdots\text{۰۰۰۱۱۱}\cdots\text{۱} \\ \bar{d} = \text{۰۰۰}\cdots\text{۰۱۱۱۱۱}\cdots\text{۱} \end{array}\right\} \Longrightarrow \bar{c} \text{ مرتب است}$$

پس در هر حالت $\bar{c}$ مرتب است.

□

مرتب‌ساز زوج و فرد

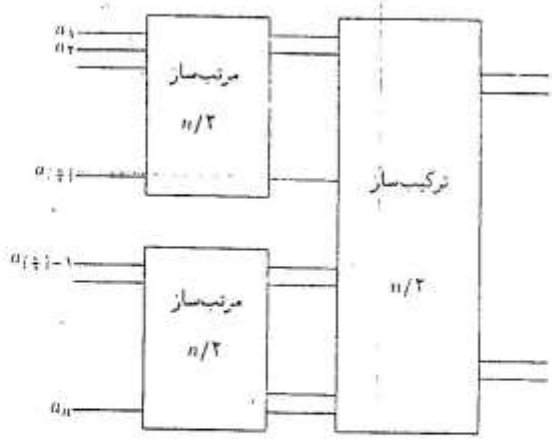
این مرتب‌ساز بر اساس ترکیب‌ساز زوج‌فرد است (شکل ۳۲.۴.)
اثبات درستی واضح است.

تحلیل ترکیب‌ساز و مرتب‌ساز زوج و فرد

$M(n)$: اندازه‌ی ترکیب‌ساز nتایی.
$S(n)$: اندازه‌ی مرتب‌ساز nتایی.
$m(n)$: عمق ترکیب‌ساز (همان زمان است.)
$s(n)$: عمق مرتب‌ساز.

$$M(n) = \begin{cases} ۱ & n = ۱ \\ ۲M(n/۲) + n - ۱ & \end{cases} \Longrightarrow M(n) = n\log_۲ n + ۱$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳۲.۴: مرتب‌ساز زوج‌فرد.

$$S(n) = 2S(n/2) + M(n/2) = \frac{n}{4}\log^2 n + O(n)$$

$$m(n) = \begin{cases} 1 & n = 1 \\ m(n/2) + 1 & \end{cases} \Longrightarrow m(n) = \log n - 1$$

$$s(n) = s(n/2) + m(n/2) = \frac{1}{2}\log n(\log n + 1)$$

بدیهی است که حد پایین برای اندازه و عمق (زمان) هر شبکه‌ی مرتب‌ساز به ترتیب $\Theta(n \log n)$ و $\Theta(\log n)$ است (چرا؟)

در سال ۱۹۸۳ Ajtai، Komlos و Szemardi موفق به ارایه‌ی شبکه بهینه‌ای با اندازه‌ی $O(n \log n)$ و عمق $O(\log n)$ شدند[۲۳]. این شبکه به نام AKS معروف است.

[۲۳] البته با ثابت بسیار بزرگ، به طوری که برای $n < 1300$ از روش زوج‌فرد بهتر می‌شود.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۴.۴ روش برنامه‌ریزی پویا

مسئله‌هایی هستند که حل آن‌ها به روش تقسیم‌وحل موجب حل تکراری زیرمسئله‌های از این مسئله می‌شود. این گونه مسئله‌ها را به‌تر است به جای حل از بالا به‌پایین و به صورت بازگشتی، از پایین به بالا حل کنیم؛ زیر مسئله‌ها فقط یک‌بار حل کرده و حاصل آن‌ها را در جدولی ذخیره کنیم، تا اگر برای حل زیرمسئله‌های بزرگ‌تر مکرراً به نتیجه‌ی حل یک زیرمسئله‌ی کوچکتر نیاز باشد، آن را بتوان بدون پرداخت هزینه‌ای از جدول به دست آورد.

این روش حل را «برنامه‌ریزی پویا[۲۴]» می‌گوییم. در مورد ویژگی‌های مسئله‌هایی که راه حل پویا دارند، در ادامه‌ی این بخش بیش‌تر صحبت خواهیم کرد. ولی قبل از آن به یک مسئله این چنینی توجه کنید.

۱.۴.۴ ترکیب m از n

می‌خواهیم با دریافت n و m، حاصل $\binom{n}{m}$ را تنها با عمل جمع انجام دهیم. برای این کار به تعریف زیر توجه کنید:

$$\binom{n}{m} = \begin{cases} 1 & \text{اگر } m = 0 \text{ یا } n = m \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{در غیر این صورت} \end{cases}$$

اگر این مسئله را از طریق تقسیم‌وحل، حل کنیم خواهیم داشت:

```
function Combination (n,m:integer):integer;
 begin
   if (n=m) or (m=0) then  return (1)
    else
       return (Combination(n-1,m)+Combination (n-1,m-1));
 end;
```

اگر $T(n,m)$ تعداد اعمال جمع این برداره باشد، داریم:

$$T(n,m) = \begin{cases} 0 & \text{اگر } m = 0 \text{ یا } n = m \\ T(n-1,m) + T(n-1,m-1) + 1 & \text{در غیر این صورت} \end{cases}$$

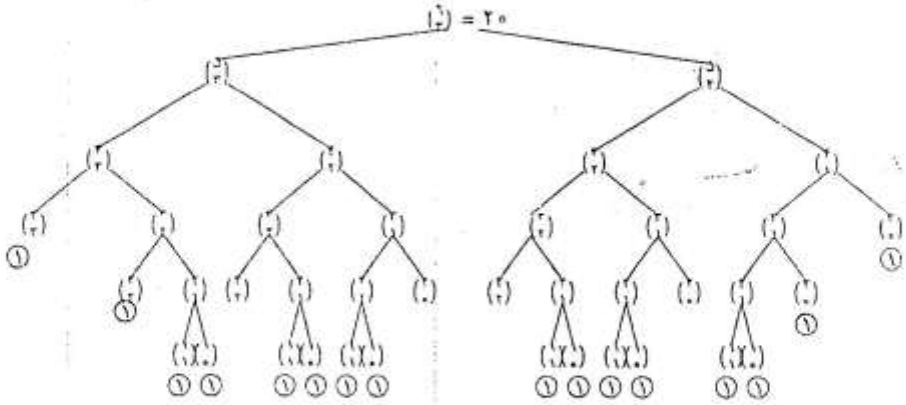
با استفاده از فرمول اصلی می‌توان دید که $T(n,m) = \binom{n}{m} - 1$. البته این جواب را از راه سریع‌تری هم می‌توان به دست آورد و آن این است که $\binom{n}{m}$ از راه فوق فقط از جمع یک‌ها به دست می‌آید. بنابراین، تعداد جمع‌ها $\binom{n}{m} - 1$ است.

این الگوریتم هرچند ساده است، ولی یک اشکال مهم دارد. اشکال الگوریتم این است که زیرمسئله‌های زیادی را چندین بار به طور مستقل حل می‌کند و حاصل آن را به دست می‌آورد. مثلاً برای محاسبه‌ی $\binom{5}{3}$، $\binom{2}{1}$ ۴ بار محاسبه می‌شود.

روش به‌تری برای حل این مسئله وجود دارد و آن این است که زیرمسئله‌ها فقط یک بار حل شوند و حاصل آن‌ها در جدولی برای استفاده‌ی مجدد ذخیره شوند. این روش حل، همان روش برنامه‌ریزی پویا است.

[۲۴] dynamic programming

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳۳.۴: درخت بازگشت برای به دست آوردن $\binom{n}{m}$.

در این روش یک ماتریس به ابعاد $(n+1) \times (m+1)$ به نام C در نظر می‌گیریم که $C(i,j)$، $0 \le i \le n$ و $0 \le j \le m$، حاوی مقدار $\binom{i}{j}$ خواهد بود. بدیهی است که قطر و ستون صفر این ماتریس برابر ۱ است. و درایه‌ی (i,j) برابر است با مجموع درایه‌های $(i-1,j)$ و $(i-1,j-1)$. الگوریتم پیدا کردن $C(n,m)$ به صورت زیر درمی‌آید:

	۰	۱	۲	۳	۴	۵	۶
۰	۱						
۱	۱	۱					
۲	۱	۲	۱				
۳	۱	۳	۳	۱			
۴	۱	۴	۶	۴	۱		
۵	۱	۵	۹	۱۰	۵	۱	
۶	۱	۶	۱۵	۱۹	۱۵	۶	۱
۷	۱	۷	۲۱	۳۴	۳۴	۲۱	۷
۸	۱	۸	۲۸	۳۵	۶۸	۵۵	۲۸
۹	۱	۹	۳۶	۶۳	۱۰۳	۱۲۳	۸۳
۱۰	۱	۱۰	۴۵	۹۹	۱۶۶	۲۲۶	۲۰۶

شکل ۳۴.۴: ماتریس برای محاسبه‌ی $\binom{n}{m}$.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
Function Combination( n,m :integer):integer;
var i,j:integer;
C: array [0..100,0..100] of integer;
begin
   for i:=1 to n do C[i,0]=1;
   for i:=1 to m do C[i,i]=1;
   for i:=2 to n do
      for j:= 1 to min(i-1, m) DO
         C[i,j]:= C[i-1,j] + C[i-1,j-1];
   return( C[n,m] );
end;
```

عناصر سطرهای این ماتریس در حقیقت همان مثلث خیام (پاسکال) هستند. شکل ۱.۴.۴، ماتریس نهایی برای محاسبه‌ی $\binom{10}{6}$ را نشان می‌دهد.

توجه کنید که در این راه حل کافی‌است به جای ماتریس از یک آرایه‌ی A[0..m] که یک سطر ماتریس را در خود دارد استفاده کنیم. البته در این صورت حلقه‌ی داخلی را باید از اندیس بالا به پایین پر کنیم، یعنی

```
Function Combination( n,m :integer):integer;
var i,j:integer;
A: array [0..100] of integer;
begin
   A[0]=1;
   for i:=2 to n do
   Begin
      if i<=m then A[i]=1;
      for j:= min(i-1, m) downto 1 DO
         A[j]:= A[j] + A[j-1];
   End;
   return(A[m]);
end;
```

تعداد اعمال جمع در این الگوریتم برابر

$$T(n,m) = ۱ + ۲ + ... + m - ۱ + m(n - m) = m(۲n - m - ۱)/۲$$

است که برای برخی مقادیر n و m کوچک‌تر از $\binom{n-1}{m-1}$ ولی برای مقادیری مانند $n = m$ بزرگتر از آن است. علت آن محاسبه‌ی درایه‌هایی از ماتریس است که نقشی در مقدار $C(n,m)$ ندارند. در واقع محاسبه‌ی درایه‌هایی که در شکل با x نشان داده شده‌اند کفایت می‌کند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
    0 1 2 3 4 . . . m
0   1
1   1 1
2   1 x 1
3   1 x x 1
4     x x x 1
.       x x x 1
.         x x x 1
.           x x x 1
.             x x x 1
.               x x x
                  x x
n                   x
```

۱ حلقه‌ی داخلی الگوریتم با این تغییر به صورت زیر در خواهد آمد.

```
Begin
   if i<=m then A[i]=1;
   for j:= min(i-1, m) downto minimum(1,m-n+i)  Do
      A[j]:= A[j] + A[j-1];
End;
```

تعداد اعمال جمع در این صورت برابر است با $(n-m)m$ که با توجه به این که $\binom{n}{m} = \binom{n}{n-m}$ این تعداد حداقل است و داریم $(n-m)m \leq \binom{n}{m} - 1$.

مسایلی که به روش برنامه‌ریزی پویا حل می‌شوند، دارای ویژگی‌های زیر هستند.

۱. مساله بهینه‌سازی است. (پارامتری کمینه یا بیشینه شود).

۲. زیر مسایل آن هم بهینه است.

۳. حل مسئله به روش تقسیم‌وحل، موجب تکرار حل زیر مسئله‌های مشابه خواهد شد. بعد از داشتن ویژگی اول و دوم چک میکنیم آیا زیر مسایل به صورت تکراری حل می‌شوند یا نه؟ اگر این گونه بود از این روش استفاده می‌کنیم.

۲.۴.۴ ضرب ماتریس‌ها

n ماتریس M_1 تا M_n داده شده‌اند. ابعاد ماتریس M_i برابر $d_{i-1} \times d_i$ است. می‌خواهیم این ماتریس‌ها را به‌صورت زیر در هم ضرب کنیم:

$$M = M_1 \times M_2 \times \cdots M_n$$

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

ترتیب انجام اعمال ضرب را طوری تعیین کنید تا تعداد کل ضرب‌های اعداد حقیقی کمینه شود. می‌دانیم که برای ضرب دو ماتریس $A \times B$ که ابعاد A و B به‌ترتیب $p \times q$ و $q \times r$ هستند به‌تعداد pqr ضرب اعداد حقیقی انجام می‌شود.

مثال

می‌خواهیم چهار ماتریس زیر را درهم ضرب کنیم (اندازه‌ی ماتریس‌ها در پایین آن‌ها آمده است):

$$M = \underset{[10 \times 20]}{M_1} \times \underset{[20 \times 50]}{M_2} \times \underset{[50 \times 1]}{M_3} \times \underset{[1 \times 100]}{M_4}$$

برای حل بازگشتی این مسئله، ابتدا زیرمسئله را تعریف کنیم. ضرب ماتریس‌های $M_{ij} = M_i \times M_{i+1} \times \cdots \times M_j$ را زیرمسئله‌ی P_{ij} و تعداد کمینه‌ی ضرب اعداد حقیقی برای آن‌را $c[i,j]$ می‌نامیم. اگر آخرین ضربی که در P_{ij} انجام می‌شود بین ماتریس M_k و M_{k+1} (برای $i \le k \le j-1$) باشد، در آن‌صورت مسئله به دو زیرمسئله‌ی P_{ik} و $P_{k+1,j}$ تبدیل می‌شود و $M_{ij} = M_{k+1,j} \times M_{ik}$. و در آن‌صورت $c[i,j] = c[i,k] + c[k+1,j] + d_i d_k d_j$. در این فرمول $d_i d_k d_j$ هزینه‌ی ضرب $M_{k+1,j} \times M_{ik}$ است.

برای پیدا کردن بهترین k باید زیرمسئله‌های فوق را به‌صورت بازگشتی برای همه‌ی kها حل کنیم و حالت کمینه را پیدا کنیم. یعنی

$$c[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \{c[i,k] + c[k+1,j] + d_i d_k d_j\} & \text{if } i < j \end{cases} \qquad (1)$$

این الگوریتم در واقع همه‌ی حالات ضرب n ماتریس را بررسی می‌کند و بین آن‌ها حالت بهینه را به‌دست می‌آورد. هزینه‌ی آن متناسب با تعداد حالات ممکن آن است که آن‌را با $T(n)$ نمایش می‌دهیم. با توضیحات بالا داریم:

$$T(n) = \begin{cases} 1 & n = 1 \\ \sum\limits_{i=1}^{n-1} T(i)T(n-i) & n > 1 \end{cases}$$

می‌توان نشان داد که $T(n) = C(n-1)$ که $C(n)$ nامین عدد کاتالان[۲۵] است که برابر است با

$$C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{\frac{3}{2}})$$

هم‌چنین روش است که راه‌حل بازگشتی تعداد زیادی زیرمسئله‌ی تکراری را حل می‌کند. این همان ویژگی است که برای حل پویا مناسب است.

۲۵ـ

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

راه‌حل پویا

این راه‌حل در واقع حل از پایین به بالای درخت بازگشت مربوط به حل بازگشتی است، که در آن نتایج زیرمسئله‌های کوچک‌تر در ماتریسی ذخیره می‌شوند تا برای محاسبه‌ی زیرمسئله‌های بزرگ‌تر مورد استفاده قرار گیرند.

ماتریس C[1..n,1..n] را تعریف می‌کنیم که C[i,j] برای $1 \le i \le j \le n$، نشان دهنده‌ی هزینه‌ی بهینه‌ی محاسبه‌ی ماتریس M_{ij} است. هم‌چنین R[i,j] اندیس k را نشان می‌دهد که حل بهینه‌ی مسئله منجر به ضرب دو زیرمسئله‌ی $M_{ik} \times M_{k+1,j}$ برای $i \le k < j$ می‌شود. اساس الگوریتم پویا همان فرمول شماره‌ی ۱ است. بنابراین الگوریتم پویا به‌صورت زیر خواهد بود:

```
for i:=1 to n do C[i,i] :=  0;
for l:= 2 to n do {number of matrices multiplied}
    for i:= 1 to n-l+1 do begin
        j:= i + l -1;
        {solving Mij}

        for k:= i to j-1 do begin
            C[i,j] := min {C[i,k] + C[k+1,j]} + d[i]*d[k]*d[j];
            R[i,j] := the value of k that makes the minimum
        end;
    end
end
```

با استفاده از ماتریس R می‌توانیم نحوه‌ی ضرب ماتریس‌ها را به‌صورت پرانتزگذاری شده به‌دست آوریم. این کار با فراخوانی رویه‌ی Print_Results(1,n) انجام می‌شود:

```
Procedure Print_Results(i,j:integer);
begin
   if i==j then
      return
   else begin
      k:= R[i,j];
      write ('(');
      Print_Results(i,k);
      write (' x ');
      Print_Results(k+1,j);
   end
```

با رویه‌ی مشابه می‌توان ماتریس حاصل را به‌دست آورد.

روشن است که زمان اجرای الگوریتم پویا از $O(n^3)$ و مقدار حافظه‌ی مصرفی آن از $O(n^2)$ است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مثال

می‌خواهیم شش ماتریس با اندازه‌های زیر را در هم ضرب کنیم:

$$\underset{[30\times 35]}{M_1} \times \underset{[35\times 15]}{M_2} \times \underset{[15\times 5]}{M_3} \times \underset{[5\times 10]}{M_4} \times \underset{[10\times 20]}{M_5} \times \underset{[20\times 25]}{M_6}$$

در انتهای الگوریتم ماتریس‌های C و R در شکل ۳۵.۴ نشان داده‌شده است. در این شکل ماتریس‌ها دوران داده شده اند و فقط بخش بالامثلثی آن‌ها نوشته شده‌اند. این شکل به‌خوبی نحوه‌ی محاسبه‌ی هزینه‌ی بهینه‌ی همه‌ی زیرمسئله‌های $M_{i,j}$ را نشان می‌دهد. مثلاً C[2,5] به‌این صورت محاسبه می‌شود.

$$C[2,5] = \min\begin{cases} C[2,2]+C[3,5]+d_1d_2d_5 = 0+2500+35\cdot 15\cdot 20 & = 13000. \\ C[2,3]+C[4,5]+d_1d_3d_5 = 2625+1000+35\cdot 5\cdot 20 & = 7125. \\ C[2,4]+C[5,5]+d_1d_4d_5 = 4375+0+35\cdot 10\cdot 20 & = 11375 \end{cases}$$
$$= 7125.$$

۳.۴.۴ مثلث بندی بهینه‌ی یک چندضلعی محدب

یک چندضلعی محدب را با دنباله‌ی $P = < v_0, v_1, \ldots, v_{n-1} >$ از رأس‌های آن در خلاف جهت عقربه‌های ساعت نشان میدهیم (شکل ؟؟) که ضلع‌های آن $< v_0, v_1 >$، $< v_1, v_2 >$، تا $< v_{n-1}, v_0 >$ هستند. چون چندضلعی محدب است، اگر v_i و v_j مجاور هم نباشند، پاره‌خط $< v_i, v_j >$، که به آن «قطر» می‌گوییم، حتماً در داخل چندضلعی قرار می‌گیرد. قطر $< v_i, v_j >$ چندضلعی را به دو چندضلعی محدب $< v_i, v_{i+1}, \ldots, v_j >$ و $< v_j, v_{j+1}, \ldots, v_i >$ تقسیم می‌کند.

منظور از مثلث‌بندی یک چند ضلعی تعیین مجموعه‌ی T از قطرهای نامتقاطع است که چندضلعی را به مثلث‌های مختلف تقسیم کند. یک n ضلعی همواره با $n-3$ قطر به $n-2$ مثلث افراز می‌شود (چرا؟)

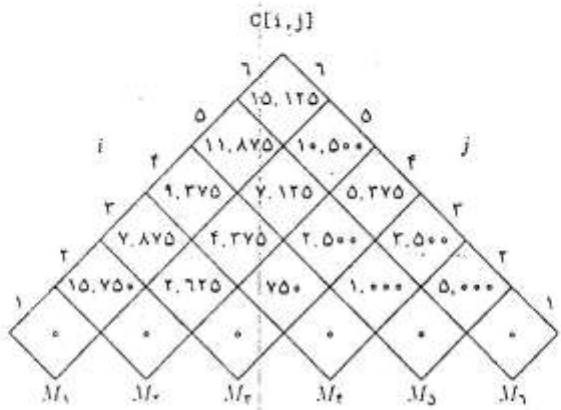
در یک چندضلعی $P = < v_0, v_1, \ldots, v_{n-1} >$ می‌توان یک تابع وزن w بر روی ضلع‌ها، قطرها یا مثلث‌های حاصل تعریف کرد. مثلث‌بندی‌یی که در آن که حاصل جمع وزن‌ها کمینه شود را مثلث‌بندی بهینه می‌گویند. طول قطرها می‌تواند یک تابع وزن باشد نمونه‌ی دیگر وزن مثلث‌هاست که بر روی هر مثلث به صورت زیر تعریف می‌شود.

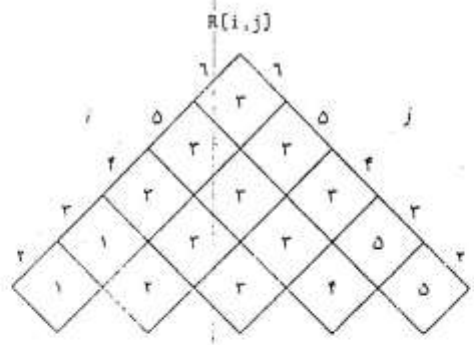
$$w(\Delta v_i v_j v_k) = \overline{v_i v_j} + \overline{v_j v_k} + \overline{v_i v_k}$$

که در آن $\overline{v_i v_j}$ فاصله‌ی اقلیدسی بین v_i و v_j است.

[illegible] یک چندضلعی محدب شباهت زیادی به [illegible] برنامه‌نویسی یک عبارت و نیز [illegible] از طریق یک درخت [illegible] قابل نمایش است. [illegible] ماتریس‌ها را می‌توان [illegible] درختی که به درخت «پارس» [illegible] برای مثال شکل ؟؟ الف ترتیب $((A_1(A_2A_3))(A_4(A_5A_6)))$ را در ضرب [illegible] یک عنصر (ماتریس) از عبارت است. اگر شاخه‌ی سمت چپ هر گره [illegible] کلی مذکور معادل ضرب $A_{i,k}A_{k,j}$ است.

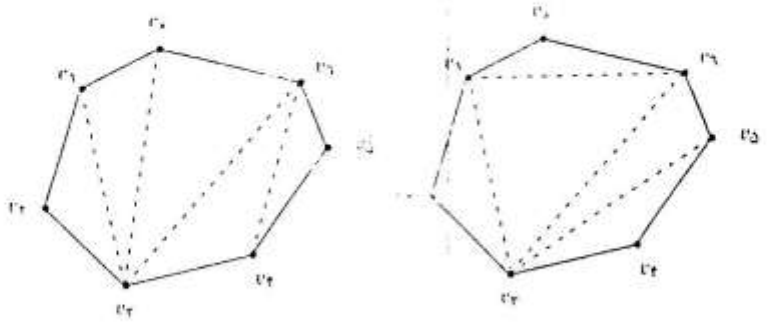
seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



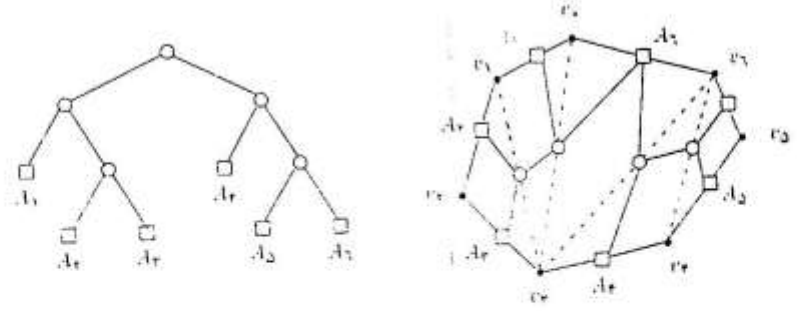


شکل ۳۵.۴: ماتریس‌های C و R برای مثال ضرب ۶ ماتریس.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳۶.۴: یک چندضلعی محدب و دو نوع مثلث‌بندی آن.



شکل ۳۷.۴: تناظر مسئله‌های ضرب ماتریس‌ها و مثلث‌بندی یک چندضلعی محدب.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

همان‌طور که ملاحظه می شود ارتباط یک به یک بین درخت پارس و عبارت پرانتزگذاری شده وجود دارد. مثلث‌بندی یک چندضلعی نیز با استفاده از درخت پارس قابل نمایش است .شکل ۳۷.۴.ب معادل بودن این مسئله‌ها را نشان می‌دهد. گره‌های داخلی درخت، قطرهای چند ضلعی هستند و ضلع $< v_0, v_6 >$ ریشه‌ی درخت محسوب می شود. برگ‌ها و گره‌های خارجی ضلع‌ها هستند. ریشه درخت یک ضلع مثلث $\Delta v_0 v_2 v_6$ در نظر گرفته شده است. این مثلث در واقع فرزندان ریشه را مشخص می‌کند که یکی از آن‌ها $< v_0, v_2 >$ و دیگری $< v_2, v_6 >$ می‌باشد. اگر دقت کنیم متوجه می‌شویم که مثلث مذکور چندضلعی اولیه را به سه قسمت تقسیم کرده است: خود مثلث و دو چندضلعی که در راس v_2 مشترکند. چند ضلعی‌های جدید که در بالا به آن‌ها اشاره شد با اضلاع اصلی چندضلعی اولیه به‌وجود آمده‌اند به‌جز ریشه آن‌ها که همان قطرهای $< v_0, v_2 >$ و $< v_2, v_6 >$ هستند. در حالت کلی چند ضلعی مثلث بندی شده با n ضلع با $n - 1$ برگ در درخت پارس ارتباط یک به یک دارد. پس حاصل‌ضرب زنجیری از n ماتریس متناظر با درخت پارس n گره‌ای است و بنابراین متناظر با چندضلعی مثلث‌بندی شده با $n + 1$ رأس است. ماتریس A_i در برنامه‌ی ضرب ماتریس‌ها متناظر با ضلع $< v_{i-1}, v_i >$ در چندضلعی $n + 1$ راسی است. قطر $< v_i, v_j >$ $(i < j)$ متناظر است با ماتریس $A_{i+1} \ldots$ در هنگام محاسبه می‌باشد. در حقیقت مسئله‌ی ضرب ماتریس‌ها حالت خاصی از مثلث بندی بهینه‌ی یک چندضلعی محدب است. یعنی هر عملیات که‌در ضرب ماتریس‌ها انجام میشود قابل اجرا در مثلث بندی است.

اگر زنجیری از ماتریس‌های $A_1 A_2 \ldots A_n$ داشته باشیم چند ضلعی با $n + 1$ راس $P = < v_0, v_1, v_2, \ldots, v_n >$ معادل آن است . اگر ماتریس A_i با ابعاد $d_{i-1} * d_i$ باشد وزن زیر را برای مثلث بندی تعریف میکنیم:

$$w(\Delta(v_i, v_j, v_k)) = d_i d_j d_k$$

مثلث‌بندی بهینه چندضلعی P با در نظر گرفتن تابع فوق درخت پارس برای پرانتزگذاری بهینه‌ی $A_1 A_2 \ldots A_n$ را ایجاد می‌کند.

هر چند حالت عکس صحت ندارد(مثلث‌بندی حالت خاص ضرب ماتریس‌ها نمی‌باشد) ولی پس از بررسی می‌بینیم که با کمی تغییرات در برنامه‌ی ضرب ماتریس‌ها می‌توانیم مثلث بندی را انجام دهیم. کافی است ابعاد ماتریس‌ها یعنی $d_0, d_1, d_2, \ldots, d_n$ را با رأس‌های $v_0, v_1, v_2, \ldots, v_n$ جابه‌جا کنیم و تمام رجوع‌ها به d تبدیل به v شوند پس:

$$\min = m[i, k] + m[k + 1, j] + w(\Delta(v_{i-1}, v_k, v_j))$$

و $m[1, n]$ ارزش مثلث‌بندی بهینه را دارد.

تعریف ساختار مثلث بندی بهینه (مرحله‌ی اول)

اگر مثلث‌بندی بهینه T با $n + 1$ راس چندضلعی $P = < v_0, v_1, v_2, \ldots, v_n >$ را در نظر بگیریم که مثلث $\Delta(v_0, v_k, v_n)$ برای k که $1 \leq k \leq n - 1$ تعریف شده باشد. ارزش T حاصل‌جمع مثلث $\Delta(v_0, v_k, v_n)$ به‌اضافه‌ی ارزش مثلث‌بندی شده‌ی زیرچندضلعی‌های بوجود آمده خواهد بود. پس مثلث‌بندی زیرچندضلعی‌های بوجود آمده باید بهینه باشد زیرا اگر چندضلعی دیگری با ارزش کمتر وجود داشته باشد با بهینه بودن T متناقض است .

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

تعریف بازگشتی (مرحله‌ی دوم)

همان‌طور که در ضرب ماتریس‌ها $m[i,j]$ را هزینه‌ی بهینه برای ضرب ماتریس‌های $A_i, A_{i+1}, \ldots, A_j$ درنظر گرفتیم پس $t[i,j]$ که $1 \le i < j \le n$ را به‌عنوان ارزش مثلث‌بندی بهینه برای چند ضلعی $< v_{i-1}, v_i, \ldots, v_j >$ منظور می‌کنیم. برای سادگی کار چندضلعی (یک ضلعی) $< v_{i-1}, v_i >$ را با ارزش صفر تعریف می‌کنیم و ارزش مثلث‌بندی بهینه‌ی P را با $t[1,n]$ مشخص می‌کنیم. پس $t[i,i] = 0$ و زمانی که $j - i \le 1$ یک چندضلعی با حداقل ۳ راس داریم که برای همه‌ی راس‌های v_k که $k = i, i+1, \ldots, j$ می‌خواهیم ارزش مثلث $\Delta(v_{i-1}v_kv_j)$ به‌اضافه‌ی چندضلعی‌های بوجود آمده‌ی $< v_{i-1}, v_i, \ldots, v_k >$ و $< v_kv_{k+1}, \ldots, v_j >$ را به حداقل برسانیم. فرمول بازگشتی عبارت است از:

$$t[i,j] = \begin{cases} 0, & i = j \\ \min_{1 \le k \le j-1} \{t[i,k] + t[k+1,j] + w(\Delta(v_{i-1}v_jv_k))\}, & i < j \end{cases}$$

همانند الگوریتم ضرب ماتریس‌ها این عملیات بازمان $O(n^3)$ و فضای $O(n^2)$ انجام می‌شود و همان کد $MatrixChainOrder$ با اندکی تغییرات قابل استفاده برای مثلث‌بندی می‌باشد.

۴.۴.۴ بزرگ‌ترین زیردنباله‌ی مشترک

زیردنباله‌ی یک دنباله‌ی داده‌شده، دنباله‌ای است که با حذف تعدادی (شاید هیچ) از عناصر آن دنباله ایجاد می‌شود. به‌عبارتی دیگر، $Z =< z_1, z_2, \ldots, z_k >$ زیردنباله‌ی $X =< x_1, x_2, \ldots, x_m >$ است اگر دنباله‌ی اکیداً صعودی $< i_1, i_2, \ldots, i_k >$ از اندیس‌های عناصر X وجود داشته باشد به‌طوری‌که برای $j = 1, \ldots, k$ داشته باشیم: $x_{i_j} = z_j$. مثلاً $Z =< B, C, D, B >$ یک زیردنباله از $X =< A, B, C, B, D, A, B >$ است که دنباله‌ی اندیس‌های مربوط $< 2, 3, 5, 7 >$ می‌باشد.

مسئله‌ی مورد نظر در این بخش پیدا کردن Z «بزرگ‌ترین زیردنباله‌ی مشترک» (Longest Common Subsequence (LCS)) برای دو دنباله‌ی داده شده‌ی X و Y است. منظور از بزرگ‌ترین طولانی‌ترین دنباله است. مثلاً اگر $X =< A, B, C, B, D, A, B >$ و $Y =< B, D, C, A, B, A >$، $< B, C, A >$ به‌طول ۳ یک دنباله‌ی مشترک برای X و Y است ولی بزرگ‌ترین آن‌ها نیست. بزرگ‌ترین زیردنباله‌ی مشترک این دو دنباله به‌طول ۴ است و $< B, C, B, A >$ یک جواب آن است.

به‌طور دقیق‌تر، ورودی دو دنباله‌ی $X =< x_1, x_2, \ldots, x_m >$ و $Y =< y_1, y_2, \ldots, y_n >$ و هدف پیداکردن LCS این دو دنباله است. اگر هر دنباله یک فایل متنی و هر عنصر آن یک سطر از آن فایل باشد، LCS این دو فایل همان فایلی است که دستور diff file1 file2 در سیستم عامل یونیکس در خروجی می‌نویسد.

زیرساختار بهینه

[illegible] برای این مسئله بررسی کلیه‌ی زیردنباله‌های X و Y و مقایسه‌ی آن‌هاست که به‌وضوح الگوریتمی [illegible] نشان می‌دهیم که این مسئله دارای زیرمسئله‌های بهینه است و راه‌حل بازگشتی آن منجر به‌حل [illegible] می‌شود. این دو ویژگی ما را به راه‌حل پویا برای این مسئله راهنمایی می‌کند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

تعریف

اگر $X = < x_1, x_2, \ldots, x_m >$ یک دنباله باشد، i امین پیشوند (prefix) X برای $i = 1..m$ را برابر $X_i \equiv < x_1, x_2, \ldots, x_m >$ تعریف می‌کنیم. مثلاً برای $X = < A, B, C, B, D, A, B >$ داریم: $X_4 = < A < B < C < B >$.

قضیه ۱۱. اگر $X = < x_1, x_2, \ldots, x_m >$ و $Y = < y_1, y_2, \ldots, y_n >$ دنباله‌های ورودی باشند و LCS این دو $Z = < z_1, z_2, \ldots, z_k >$ باشد داریم:

۱. اگر $x_m = y_n$، داریم $z_k = x_m = y_n$ و Z_{k-1} برابر LCS X_{m-1} و Y_{n-1} است.

۲. اگر $x_m \neq y_n$، آن‌گاه از $z_k \neq x_m$ نتیجه می‌گیریم که Z برابر LCS X_{m-1} و Y است.

۳. اگر $x_m \neq y_n$، آن‌گاه از $z_k \neq y_n$ نتیجه می‌گیریم که Z برابر LCS Y_{n-1} و X است.

اثبات.

۱. اگر این چنین نباشد، یعنی $z_k \neq x_m$، می‌توانیم با اضافه‌کردن $x_m = y_n$ به‌عنوان آخرین عنصر Z، یک زیردنباله‌ی مشترک بزرگ‌تر از قبل ایجاد کنیم که با این فرض که Z برابر LCS دو دنباله است تناقض دارد.

۲. اگر $z_k \neq x_m$ پس Z برابر LCS X_{m-1} و Y است. اگر Z برابر LCS X_{m-1} و Y باشد، نمی‌تواند طولش از k بیش‌تر باشد، چرا که با فرض LCS بودن Z تناقض دارد.

۳. مشابه‌ی بند (۲).

□

راه‌حل بازگشتی

از قضیه‌ی ۱۱ می‌توان دید که راه‌حل بازگشتی مسئله منجر به یک یا دو فراخوانی بازگشتی خواهد شد. اگر $x_m = y_n$، باید LCS را برای X_{m-1} و Y_{n-1} پیدا کرد و به آن x_m را اضافه کرد. اگر $x_m \neq y_n$، دو فراخوانی بازگشتی برای حل زیرمسئله‌های به‌دست آوردن LCS برای X_{m-1} و Y و نیز برای Y_{n-1} و X انجام داد. بزرگ‌ترین زیردنباله‌های این دو زیرمسئله LCS پاسخ است.

روشن است که راه‌حل بازگشتی منجر به حل تکراری زیرمسئله‌های مختلف می‌شود. بنابراین راه‌حل پویا برای این مسئله مناسب است. اگر $c[i, j]$ طول LCS برای X_i و Y_j باشد، فرمول ۲ ارتباط این مقادیر را برای زیرمسئله‌های مختلف نشان می‌دهد.

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j, \end{cases} \qquad (2)$$

راه‌حل پویا

براساس فرمول ۲ می‌توان الگوریتم پویای زیر را برای این مسئله ارائه داد. برای این کار نیاز به ماتریس c[0..m,0..n] داریم که c[i,j] طول LCS برای X_i و Y_j را در خود ذخیره می‌کند. مانند بقیه راه‌حل‌های پویا

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

برای ساختن LCS ماتریس b[1..m,1..n] را تعریف می‌کنیم که b[i,j] به‌صورت نمادین حاوی برداری است که نشان می‌دهد که c[i,j] از کدام‌یک از زیرمسئله‌ها ساخته شده است. اگر b[i,j]=↖، در آن‌صورت c[i,j] از c[i-1,j-1] ساخته شده است. اگر b[i,j]=↑، c[i,j] از c[i-1,j] و اگر b[i,j]=←، c[i,j] از c[i,j-1] به‌دست آمده است.

```
for i := 1 to m do c[i,0] := 0;
for j := 1 to m do c[0,j] := 0;
for i := 1 to m do
    for j := 1 to n do
        if x_i = y_j then begin
            c[i,j] := c[i-1,j-1] + 1;
            b[i,j] := '↖'
        end
        else if c[i-1,j] ≥ c[i,j-1]
            then begin
                c[i,j] := c[i-1,j];
                b[i,j] := '↑'
            end
            else begin
                c[i,j] := c[i,j-1];
                b[i,j] := '←'
            end
```

مثال

اگر $X = < A, B, C, B, D, A, B >$ و $Y = < B, D, C, A, B, A >$ ماتریس‌های حاصل در انتهای اجرای الگوریتم پویا مطابق شکل ۳۸.۴ خواهد بود.

روشن است که الگوریتم ارائه شده از مرتبه‌ی زمانی $O(mn)$ و میزان حافظه‌ی مصرفی آن نیز $O(mn)$ است.

ساختن LCS

با توجه به اطلاعات ذخیره‌شده در ماتریس b می‌توان LCS را با فراخوانی Print-LCS(b, X, m, n) ساخت. این رویه به‌صورت زیر است:

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

i	j	۰	۱	۲	۳	۴	۵	۶
		y_j	B	D	C	A	B	A
۰	x_i	۰	۰	۰	۰	۰	۰	۰
۱	A	۰	↑ ۰	↑ ۰	↑ ۰	↖ ۱	← ۱	↖ ۱
۲	B	۰	↖ ۱	← ۱	← ۱	↑ ۱	↖ ۲	← ۲
۳	C	۰	↑ ۱	↑ ۱	↖ ۲	← ۲	↑ ۲	↑ ۲
۴	B	۰	↖ ۱	↑ ۱	↑ ۲	↑ ۲	↖ ۳	← ۳
۵	D	۰	↑ ۱	↖ ۲	↑ ۲	↑ ۲	↑ ۳	↑ ۳
۶	A	۰	↑ ۱	↑ ۲	↑ ۲	↖ ۳	↑ ۳	↖ ۴
۷	B	۰	↖ ۱	↑ ۲	↑ ۲	↑ ۳	↖ ۴	↑ ۴

شکل ۳۸.۴: ماتریس‌های c و b در پیداکردن LCS برای $X = < A, B, C, B, D, A, B >$ و $Y = < B, D, C, A, B, A >$

```
PRINT-LCS(b, X, i, j)
if (i = 0 or j = 0)
  then return;
if b[i, j] = '↖'
  then begin
      PRINT-LCS (b, X, i − 1, j − 1);
      print x_i
  end
  else if b[i, j] = '↑'
      then PRINT-LCS (b, X, i − 1, j)
      else PRINT-LCS (b, X, i − 1, j)
```

این الگوریتم از مرتبه‌ی $O(m + n)$ است.

۵.۴.۴ درخت دودویی جست‌وجوی بهینه (Optimal BST)

درخت جست‌وجو (Binary Search Tree) یک ساختمان داده‌ی مناسب برای پیاده‌سازی فرهنگ‌های داده‌ای است. برای فرهنگی با n عنصر، اعمال درج، حذف و جست‌وجو را می‌توان با متوسط $O(\log n)$ انجام داد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

اگر داده‌ها از پیش آماده باشند، می‌توانیم آن‌ها را به‌گونه‌ای در درخت قرار دهیم، تا درختی متوازن به‌دست آید. بدین ترتیب می‌توان اعمال درج، حذف و جست‌وجو را در بدترین حالت با $O(\log n)$ انجام داد. برای ساختن چنین درختی، باید عنصر میانه را در ریشه قرار دهیم. بدین ترتیب نیمی از عناصر در زیر درخت چپ و نیمی دیگر در زیر درخت راست قرار می‌گیرند. سپس همین روش را روی زیر درخت‌های چپ و راست انجام می‌دهیم.

یک درخت دودویی جست‌وجویی را در نظر بگیرید که عمل جست‌وجو در مقایسه با بقیه‌ی اعمال بسیار زیاد اتفاق می‌افتد. مثلاً اگر اطلاعات مربوط به یک کتابخانه را با درخت دودویی جست‌وجویی پیاده‌سازی کنیم، کار عمده بر روی آن عمل جست‌وجو است و این جست‌وجوها می‌توانند «موفق» یا «ناموفق» باشد؛ یعنی کتاب در کتابخانه موجود باشد یا خیر. نکته‌ی مهم در این‌جا آن است که در عمل تعداد تقاضاهای جست‌وجو برای عناصر مختلف این درخت (کتاب‌های مختلف) بسیار متفاوت است. با توجه به آن که زمان انجام جست‌وجو برای یک عنصر متناسب با عمق آن عنصر در درخت است، اگر بتوانیم عناصری را که تعداد جست‌وجو برای آن‌ها بیشتر است به ریشه نزدیک‌تر کنیم و آن‌هایی که کمتر مورد جست‌وجو قرار می‌گیرند را دورتر از ریشه قرار دهیم، متوسط زمان جست‌وجو کمتر از حتا حالت متوازن خواهد شد. این مطلب، ایدهٔ اصلی درخت دودویی جست‌وجوی بهینه است.

در شکل ؟؟ یک درخت دودویی جست‌وجوی متوازن برای عناصر $a_1 < a_2 < \cdots < a_7$ دیده می‌شود. فرض کنید $P(a)$ احتمال آن است که یک جست‌وجوی دل‌خواه برای عنصر a باشد. و فرض کنید که

$$P(a_1) = \frac{1}{26}, \quad P(a_2) = \frac{3}{26}, \quad P(a_3) = \frac{2}{26}$$
$$P(a_4) = \frac{4}{26}, \quad P(a_5) = \frac{6}{26}$$
$$P(a_6) = \frac{3}{26}, \quad P(a_7) = \frac{7}{26}$$

با توجه به آن‌که زمان جست‌وجو برای هر عنصر متناسب است با عمق آن عنصر در درخت، متوسط زمان جست‌وجو در این درخت برابر است با:

$$\frac{1}{26}[4 \cdot 1 + (3+3) \cdot 2 + (1+2+6+7) \cdot 3] = \frac{64}{26}$$

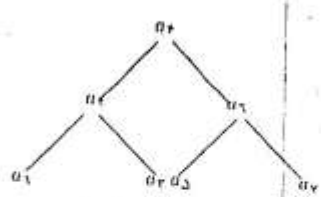
اگر به‌جای این درخت، درخت شکل ؟؟ را برای این عناصر ایجاد کنیم، با این که ارتفاع درخت بیش‌تر شده است، متوسط زمان جست‌وجو کم‌تر می‌شود:

$$\frac{1}{26}[6 \cdot 1 + (4+7) \cdot 2 + (3-3) \cdot 3 + (1+2) \cdot 4] = \frac{58}{26}$$

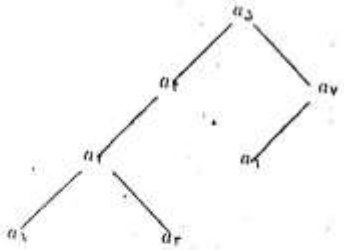
به عنوان مثالی دیگر، توجه کنید که هر کلمه‌ی استفاده‌شده در متن یک زبان برنامه‌نویسی با کلمات موجود در جدول نماد (symbol table) آن کامپایلر جست‌وجو می‌شود و این کار به دفعات زیادی انجام می‌شود. مثلاً در یک زبان پاسکال، ممکن است یک جدول نماد از ۵ کلمه‌ی کلیدی به‌صورت شکل ۴۰.۴ ساخته شود.

جست‌وجوهای ناموفق برای پیداکردن عناصری که در درخت نیستند هم در شکل بهینه‌ی درخت تأثیر می‌گذارند و باید آن‌ها را در نظر گرفت. این جست‌وجوها را در یک درخت دودویی جست‌وجو می‌توان با عناصر خارجی آن نشان داد. به‌ازای هر اشاره‌گر Nil در درخت یک عنصر خارجی قرار می‌دهیم. مثلاً در شکل ۴۰.۴ اگر عناصر خارجی را b_i بنامیم، b_0 نشان‌دهنده‌ی جست‌وجو برای کلیه‌ی کلمات کوچک‌تر از begin است، b_1 نشان‌دهنده‌ی جست‌وجو برای کلیه‌ی کلماتی است که از begin بزرگ‌تر و از else کم‌تر هستند.

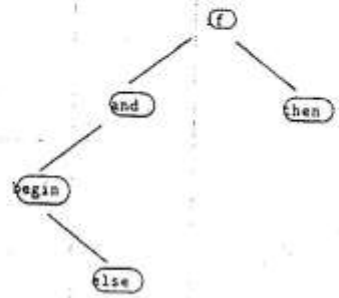
seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۳۹.۴: درخت متوازن با متوسط زمان جست‌وجوی $\frac{32}{28}$.



شکل ۴۰.۴: درخت نامتوازن با متوسط زمان جست‌وجوی $\frac{58}{28}$.



شکل ۴۰.۴: یک درخت دودویی جست‌وجو برای کلمات کلیدی.

مساله: ثابت کنید تعداد عناصر خارجی، برای یک درخت دودویی جست‌وجو با n عنصر، $n + 1$ است.

برهان:

۱. استفاده از روش استقرایی

۲. به‌ازای هر عنصر غیر از ریشه، یک پدر و در نتیجه یک یال تناظر وجود دارد. (در کل $n - 1$ یال) از طرفی دیگر هر عنصر دو فرزند دارد که برخی از فرزندها عناصر خارجی هستند. بنابراین $2n$ یال به عناصر داخلی و خارجی وارد می‌شوند. پس $2n - (n - 1) = n + 1$ عنصر به عنوان عنصر خارجی باقی می‌ماند.

۳. اگر n عنصر درخت $a_1 < a_2 < \cdots < a_n$ باشند، آن‌گاه $n + 1$ بازه بین آن‌ها برای عناصر خارجی b_i موجود است:

$$b_0 < a_1 < b_1 < a_2 < \cdots < a_n < b_n$$

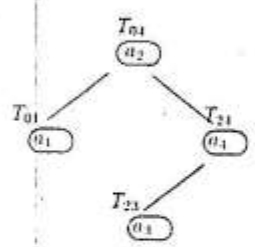
ساختن درخت دودویی جست‌وجوی بهینه

فرض کنید که

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۰	$c_{۰۰} = ۰$ $w_{۰۰} = ۲$	$c_{۰۱} = ۹$ $w_{۰۱} = ۹$ $r_{۰۱} = a_۱$	$c_{۰۲} = ۱۸$ $w_{۰۲} = ۱۲$ $r_{۰۲} = a_۱$	$c_{۰۳} = ۲۵$ $w_{۰۳} = ۱۴$ $r_{۰۳} = a_۱$	$c_{۰۴} = ۳۲$ $w_{۰۴} = ۱۶$ $r_{۰۴} = a_۲$
۱		$c_{۱۱} = ۰$ $w_{۱۱} = ۳$	$c_{۱۲} = ۶$ $w_{۱۲} = ۶$ $r_{۱۲} = a_۲$	$c_{۱۳} = ۱۱$ $w_{۱۳} = ۸$ $r_{۱۳} = a_۲$	$c_{۱۴} = ۱۸$ $w_{۱۴} = ۱۰$ $r_{۱۴} = a_۲$
۲			$c_{۲۲} = ۰$ $w_{۲۲} = ۱$	$c_{۲۳} = ۳$ $w_{۲۳} = ۳$ $r_{۲۳} = a_۳$	$c_{۲۴} = ۸$ $w_{۲۴} = ۵$ $r_{۲۴} = a_۴$
۳				$c_{۳۳} = ۰$ $w_{۳۳} = ۱$	$c_{۳۴} = ۳$ $w_{۳۴} = ۳$ $r_{۳۴} = a_۴$
۴					$c_{۴۴} = ۰$ $w_{۴۴} = ۱$

جدول ۱.۴: مقادیر c_{ij}، w_{ij}، r_{ij} برای مثال.



شکل ۴۲.۴: درخت دودویی جست‌وجوی بهینه برای مثال.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۶.۴.۴ مسئله‌ی کوله پشتی (Knapsack Problem)

کوله‌پشتی از مسئله‌های کلاسیک و مهم و با کاربرد زیاد است که می‌تواند به این صورت مطرح شود: دزدی به منظور سرقت به فروشگاهی می‌رود. در فروشگاه تعدادی بسته از اجناس مختلف وجود دارد. هر جنس ارزش مشخصی دارد و وزن آن نیز معلوم است. دزد برای برداشتن اجناس یک کوله پشتی به همراه دارد که با آن می‌تواند تا حداکثر M کیلوگرم را حمل کند (وزن‌ها همگی اعداد صحیح هستند.) به فرض این که دزد نمی‌تواند قسمتی از یک بسته را بردارد. الگوریتمی طراحی کنید که مشخص کند که دزد باید کدامیک از بسته‌ها را بردارد که مجموع وزن آن‌ها از M کیلوگرم بیشتر نشود و مجموع ارزش آن‌ها نیز بیشترین مقدار ممکن شود. باتوجه به این که در این مسئله‌ی برداشتن جزئی از یک بسته مجاز نیست و دزد تنها می‌تواند یک بسته را به تمامی بردارد و یا اصلاً آن را برندارد، به این مسئله، مسئله کوله پشتی صفر و یک (O-1 Knapsack) نیز می‌گویند. می‌خواهیم ببینیم که آیا برای این مسئله راه حل سریع وجود دارد.

مسئله را می‌توان در حالت کلی به‌صورت زیر تعریف کرد:

ورودی:

- یک کوله‌پشتی با توانایی حمل M واحد وزن جنس
- N نوع جنس
- تعداد بسته‌ی از بار نوع iام برابر N_i (یا `Number[i]` در برنامه‌ها)
- وزن جنس نوع iام W_i (یا `Weight[i]` در برنامه‌ها). فرض می‌کنیم که $W_i \leq M$.
- ارزش جنس نوع iام C_i (یا `Cost[i]` در برنامه‌ها)

فرض: M و W_i اعداد صحیح هستند.

می خواهیم کوله‌پشتی را با این بارها کاملاً یا تا حد امکان پر کنیم به‌طوری‌که اگر بارها ارزش داشته باشند، مجموع ارزش بارها بیشینه شود.

حالت اول:

$N_i = 1$ ،$C_i = 0$ و کوله‌پشتی پر شود.

این مسئله یک مسئله‌ی NP−تمام است و راهی به‌جز بررسی تمام حالت‌ها ندارد ولی باید با نظم خاصی این حالت‌ها را بررسی کنیم. برای این منظور ابتدا از روش پس‌گرد (Backtracking) استفاده می‌کنیم.

راه‌حل پس‌گرد

بار شماره‌ی i را در نظر بگیرید. این بار دو حالت ممکن است داشته باشد. یا جزء انتخاب‌هاست که در آن صورت وزن آن را از توانایی حمل کوله‌پشتی کم می‌کنیم. یا جزء انتخاب‌ها نیست که در آن صورت بارهای بعدی را امتحان می‌کنیم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

برای حل این مسئله تابع Knapsack را برای پرکردن یک کوله‌پشتی به توانایی حمل W از جنس‌های شماره‌ی i تا N می‌نویسیم. این تابع در صورتی‌که مسئله جواب داشته باشد True و در غیر این صورت False برمی‌گرداند.

```
function Knapsack(W, i: Integer): Boolean;
begin
 if W = 0 then return True;
 if (W < 0) or (i > N) then return False;
 { load #i is a candidate }
 if Knapsack(W - Weight[i], i + 1) then
 begin
   WriteLn(i, Weight[i]);
   return True;
 end
 else return (Knapsack(W, i + 1));
end;
```

بدترین حالت هنگامی است که مسئله جواب نداشته باشد. در آن‌صورت الگوریتم از مرتبه‌ی $\Theta(2^n)$ است.

$$T(n) = 2T(n-1) + \Theta(1) \Rightarrow T(n) = \Theta(2^n)$$

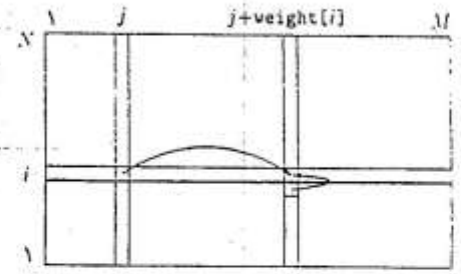
راه حل کورکورانه

راه حل کورکورانه برای مسئله‌ی فوق بررسی تمام حالات ممکن به ترتیب زیر است: چنان‌چه بخواهیم یکی از وزنه‌ها را انتخاب کنیم به جای آن ۱ و در غیر این صورت به جای آن صفر در نظر می‌گیریم. بنابراین 2^N حالت مختلف خواهیم داشت که هر حالت نمایش دودویی اعداد صفر تا $2^N - 1$ می‌باشد. بنابراین می‌توانیم با یک حلقه تمام حالت‌های ممکن را آزمایش کنیم:

```
for i := 0 to 2^N - 1 do
begin
  find the N-bit binary representation of i;
  find the loads  with 1's in binary form of i;
  if the sum of weights of the selected loads
      is equal to M then this is one solution
end
```

در بدترین حالت این دو الگوریتم مانند هم عمل می‌کنند. و با توجه به NP–تمام بودن مسئله این امر قابل انتظار است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۳.۴: نحوه‌ی پرکردن ماتریس S در الگوریتم پویا.

راه‌حل پویا

این مسئله به‌وضوح دارای خصوصیت زیرساختار بهینه است: هرگاه به اندازه ی m درون کوله‌پشتی جنس قرار داده باشیم، پس از این مسئله تبدیل می‌شود به پرکردن یک کوله‌پشتی با ظرفیت $M - m$ با بسته‌هایی که باقی مانده‌اند. واضح است که هر جواب بهینه برای مسئله‌ی اصلی، شامل یک جواب بهینه برای این مسئله نیز هست.

ماتریس $S[M \times N]$ را به صورت زیر تعریف می‌کنیم: `S[i, j]` نشان می‌دهد که آیا می‌توان کوله‌پشتی با اندازه‌ی j را با بارهای از ۱ تا i پر کرد یا خیر. اگر نتوان، این مقدار صفر است و در غیر این صورت برابر $k > 0$ است که k شماره‌ی آخرین باری است که در کوله‌پشتی قرار می‌دهیم.

در حالت کلی داریم:

`S[i, j] = S[i - 1, j]` یا `S[i - 1, j - weight[i]]`

شکل ۴۳.۴ نحوه‌ی پرکردن ماتریس را نشان می‌دهد. این الگوریتم به شرح زیر است: (فرض $k = j + weight[i]$)

- در سطر اول داریم: $S[1, weight[1]] = 1$
- سطر i بر اساس سطر $i - 1$ پر می‌شود.
- ابتدا همه‌ی مقادیر سطر $i - 1$ در سطر i کپی می‌شود.
- $S[i, k]$ را از $S[i, j]$ پر می‌کنیم، در صورتی که $S[i, k]$ توسط بارهای قبلی پر نشده باشد.
- اگر $j = 0$ یا $S[i, j] <> i$، $S[i, k]$ را فقط با یک عدد بار i پر می‌کنیم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
for i := 1 to N do
   for j:= 1 to M do S[i,j] := 0;
for i := 1 to N do
  for j := 0 to M - Weight[i] do
  begin
    if i > 1 then S[i, j] := S[i-1, j];
    k:=j + weight[i];
    if (j=0 or (S[i,j] > 0))
    then {we may be able to fill S[i,k]}
    if (k <= M) and (S[i,k] =0)
    then S[i,k] := i;
  end;
```

با یک آرایه همین کار را می‌توان انجام داد.

```
for j:= 1 to M do S[j] := 0;
for i := 1 to N do
  for j := 0 to M - Weight[i] do
  begin
    k:=j + weight[i];
    if (j=0 or ((S[j] > 0) and (S[j] <> i))
         %load i can be used only once!
    then {we may be able to fill S[k]}
    if (k <= M) and (S[k] =0)
    then S[k] := i;
  end;
```

و یک جواب را هم می‌توان نوشت:

```
procedure print_results (j:integer);
{print the one solution for knapsack of size j
 we assume that this knapsack can be filled}
begin
     if j = 0 then return;
(*)  k := S[j]; {k has to be > 0}
     writeln (k);
     print_results (j-k)
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

به‌دلیل این که اگر مسئله‌ی S[i,j] بیش از یک جواب داشته باشد (یکی با استفاده از بار i و دیگری با بارهای کمتر از آن) ما اولویت را به جواب S[i-1,j] می‌دهیم، راه‌حل فوق درست است. یعنی اگر $S[j] = k$، حتماً $S[j-k] < k$ یا $j = k$.

حالت دوم (کلی):

- $C_i = 0$، $1 < n_i < \infty$، و کوله‌پشتی کاملاً پر شود.

در این حالت درایه‌ی $S[j]$ شامل دو مولفه است:

- Last: شماره‌ی آخرین باری که در کوله‌پشتی j قرار می‌گیرد. اگر این کوله‌پشتی را نتوان پر کرد، این مولفه برابر صفر است.
- No: تعداد استفاده شده از بار $S[j].last$ در این کوله‌پشتی.

$Number[i]$: تعداد بار از نوع i.

شرح الگوریتم:

برای هر i سطر iام ماتریس را پر می‌کنیم. در این صورت

- برای $j = 0...M - Weight[i]$، از $S[j]$ تحت شرایطی $S[j + weight[i]]$ را با قرار دادن یک عدد دیگر از بار iام پر می‌کنیم.
- $S[j + weight[i]]$ فقط در صورتی که قبلاً پر نشده باشد، پر می‌شود.

[illegible] می‌شود اگر $0 \le j$ [illegible] $S[j]$ [illegible] آن [illegible] i نباشد.

```
for j := 1 to M do S[j].last := 0;
for i := 1 to N do
  for j := 0 to M - weight[i] do
  begin
    k := j+weight[i]; {trying to fill S[k]}
    {conditions to use load i for the first time}
    if (j=0) or ((S[j].last > 0) and (S[j].last <> i))
    then
       if (k <= M) and (S[k].last = 0)
       then begin
           S[k].last := i;
           S[k].No := 1
       end
    else
    {conditions to use more than one load i}
    if (S[j].last = i) and (S[j].No < Number[i])
    then
       if (k <= M) and (S[k].last = 0)
       then begin
           S[k].last := i;
           Inc (S[k].No)
       end
end;
```

رویه‌ی print_results قبلی با تغییر دستور (*) به k := S[j].last در این حالت درست کار می‌کند.

حالت سوم:

$C_i = 0$، $n_i = \infty$، و کوله‌پشتی کاملاً پر شود.
در این حالت بار نوع i می‌تواند بیش از یک بار استفاده شود.

```
S[0] := 0;
for i := 1 to N do
  for j := 0 to M do
  begin
    k:= j + weight [i];
    if (j = 0 or S[j] <> 0) then
       if k <= M then S[k] := i;
  end;
```

رویه‌ی print_results در این حالت نیز درست کار می‌کند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مسئله‌ی خرد کردن پول (اگر نخواهیم تعداد سکه‌ها کمینه باشد) یک مسئله‌ی کوله‌پشتی با $n_i = \infty$ است.

حالت چهارم:

$n_i = 1$ و $C_i > 0$ باشد و کوله پشتی کاملاً پر شود.

در این حالت یک مولفه‌ی جدید برای درایه‌ی $S[i,j]$ در نظر می‌گیریم:

$S[i,j].value$: بیشترین ارزش بارهای از نوع ۱ تا i برای پرکردن کوله‌پشتی به‌اندازه‌ی j.

```
for i := 1 to N do
   for j:= 1 to M do S[i,j].value := 0;
for i := 1 to N do
  for j := 0 to M - Weight[i] do
  begin
    if i > 1 then S[i, j] := S[i-1, j];
    k := j + weight[i];
    if (j = 0 or (S[i-1,j].value > 0))
    then {we may be able to fill S[i,k]}
    if (k<=M) and (S[i,k].value < S[i-1,j].value + Cost[i])
    then begin
          S[i,k].last := i;
          S[i,k].value := S[i-1,j].value + Cost [i]
    end
  end;
```

جواب را نیز می‌توان نوشت:

```
procedure print_results (i, j: integer);
begin
    if j = 0 then return;
    k := S[i,j].last; {k has to be > 0}
    writeln (k);
    print_results (k-1, j-k)
```

مسئله را می‌توان با یک آرایه هم به‌صورت زیر حل کرد. ولی جنس‌های استفاده شده را نمی‌توان به‌دست آورد. در این صورت، آرایه‌ی $S[0...M]$ همان ارزش است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
for j:= 0 to M do S[j] := 0;
for i := 1 to N do
  for j := M downto weight[i] do
  begin
    if i = 1 then S[weight[i]] := Cost[i];
    else begin
       k := j - weight[i];
       if S[j] < S[k] + Cost[i] then
           S[j] := S[k] + Cost[i]
    end
  end;
```

برای پیدا کردن با ارزش‌ترین جواب ممکن است کوله‌پشتی کاملاً پر نشود. در این صورت در سطر N ام دنبال بزرگ‌ترین ارزش می‌گردیم.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۵.۴ روش حریصانه درطراحی الگوریتم‌ها

روش حریصانه (Greedy) یکی از روش‌های ساده، جالب و سریع برای حل دقیق مسئله‌هاست. البته باید مسئله دارای شرائط خاصی باشد تا بتوان از این روش برای حل آن استفاده کرد. متأسفانه تعداد کمی از مسئله‌ها راه‌حل حریصانه دارند. از جمله‌ی آن‌ها مسئله‌های مهمی هستند که راه‌حل تعدادی از آن‌ها به‌دلیل نوآوری قابل توجه به‌نام طراح‌های آن‌ها ثبت شده‌اند: مانند الگوریتم هافمن (Huffman) برای پیدا کردن کدهای بهینه برای فشرده‌سازی متن‌ها، الگوریتم دایکسترا (Dijkstra) برای پیدا کردن همه‌ی کوتاه‌ترین مسیرها از یک رأس در یک گراف، و الگوریتم‌های پریم (Prim) و کروسکال (Kruskal) برای پیدا کردن درخت پوشای کمینه در یک گراف، و الگوریتم Fleury برای پیداکردن یک دور اولری در گراف.

روش حریصانه اگر منجر به پیدا کردن جواب دقیق مسئله نشود یک روش تقریبی برای حل آن مسئله است که در صورتی که بتوان اثبات کرد که جواب به‌دست آمده در بدترین حالت چه‌مقدار با جواب دقیق فاصله دارد، بسیار با ارزش است و یکی از روش‌های مهم در طراحی الگوریتم‌های تقریبی «قابل اثبات» است. درصورتی‌که چنین اثباتی وجود نداشته باشد، روش حریصانه در واقع یک روش مکاشفه‌ای (heuristic) برای حل مسئله است که ممکن است در عمل کاربر فراوان داشته‌باشد ولی از نظر علمی چندان با ارزش نیست.

مسائلی که با روش حریصانه قابل حل هستند دارای خصوصیات زیر می‌باشند:

- مسئله بهینه‌سازی (optimization) است.
- برای حل بهینه‌ی مسئله باید زیرمسئله‌های آن‌را نیز به‌صورت بهینه حل کرد (optimal subproblem). مسئله‌هایی که به‌روش پویا قابل حل بودند نیز دو ویژگی فوق را داشتند.
- انتخاب حریصانه (greedy choice) دراین‌گونه مسائل بهترین انتخاب است و عوض نمی‌شود. (تفاوت با روش پویا در این ویژگی است.)

به‌طور کلی می‌توان یک الگوریتم حریصانه را مطابق زیر بیان کرد:

```
function greedy (C: set): set
(C: the set of  all candidates)
begin
  Makenull(S); (S is the solution)
  while not solution(S) and not Empty(C) do
  begin
     x := an element in C maximizing select(x);
     C := C - {x}
     if feasible (S U {x}) then S:= S U {x}
  end
  if solution(S) then return (S)
                 else return ("NO solution")
end;
```


seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱.۵.۴ ویژگی‌های انتخاب حریصانه

انتخاب نامزد درصورت نداشتن تضاد با انتخاب‌های انجام‌شده انتخاب نهایی است و عمل پس‌گرد (backtracking) برای تغییر انتخاب صورت نمی‌گیرد. این انتخاب براساس تابع انتخاب (selection function) و براساس مقادیر محلی صورت می‌گیرد. در واقع الگوریتم حریصانه به‌گونه‌ای است که انتخاب مبتنی بر بهینه‌سازی محلی (local optimization) منجر به بهینه‌سازی سراسری (global optimization) می‌شود که هدف مسئله است.

۲.۵.۴ انتخاب فعالیت‌ها (Activity Selection Problems)

N فعالیت مختلف $t_1, t_2, \ldots, t_n$ داده شده‌اند که همگی از یک منبع غیرقابل اشتراک استفاده می‌کنند. برای هر فعالیت دو پارامتر زمان شروع و زمان پایان مشخص شده است. هدف پیدا کردن بیشترین تعداد این فعالیت‌هاست که بتوانند از منبع استفاده کنند.

ورودی: برای هر فعالیت t_i (برای $1 \le i \le n$) مقادیر زیر داده شده‌اند:

- زمان شروع: s_i
- زمان پایان: f_i

هدف: انتخاب حداکثر فعالیت های ممکن.

مثال: هر فعالیت می‌تواند یک تقاضا برای استفاده از مثلاً سالن اجتماعات یک دانشگاه باشد.

ارائه‌ی یک روش ساده برای حل مساله:

می توان برای هرفعالیت موجود، تمام فعالیت‌های دیگر که با آن در تضاد هستند را حذف نموده و این روش را تکرار کرد و کلیه‌ی مجموعه فعالیت‌های ممکن را به‌دست آورد. در پایان بزرگ‌ترین مجموعه‌ی ممکن جواب است. الگوریتم بسیار کند و از مرتبه‌ی $O(n!)$ است چون همه‌ی جای‌گشت‌ها را بررسی می‌کند.

روش حریصانه

در الگوریتم حریصانه به‌دنبال روشی برای انتخاب یک فعالیت مناسب هستیم که در صورتی‌که با انتخاب‌های تناقضی نداشته باشد آن‌را به‌صورت نهایی انتخاب کنیم. برای این کار می‌توانیم ترتیب انتخاب‌ها را براساس ترتیب اولیه، طول فعالیت‌ها، زمان شروع، یا زمان پایان (از کوچک به بزرگ یا برعکس) انجام دهیم.

مشخص است که در دو مورد اول به مشکل برمی‌خوریم، مثلاً در مورد ترتیب انتخاب براساس طول فعالیت‌ها، اگر از سه فعالیت، کوچک‌ترین آن‌ها با دو فعالیت دیگر (که از هم مجزا هستند) اشتراک داشته باشد، ما فقط همان کوچک‌ترین فعالیت را می‌توانیم انتخاب کنیم، در صورتی‌که دو فعالیت دیگر جواب بهینه است. برای ترتیب‌های دیگر، به مر آخرین ترتیب نیز می‌توان مثال نقض پیدا کرد.

بنابراین، به نظر می‌رسد که اگر فعالیت‌ها را به‌ترتیب پایان زمان‌شان انتخاب کنیم و پس از انتخاب یک فعالیت، همه‌ی فعالیت‌های متضاد با آن را حذف کنیم، جواب بهینه به‌دست می‌آید. البته درستی این الگوریتم حریصانه را باید اثبات کنیم.

مراحل کلی اثبات درستی یک الگوریتم حریصانه:

۱. اثبات می‌کنیم که یک راه حل بهینه وجود دارد که شامل اولین انتخاب الگوریتم پیشنهادی است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۲. با حذف انتخاب اولیه و انتخاب‌های متضاد با آن یک زیرمسئله به‌دست می‌آید که باید آن‌را نیز به‌صورت بهینه و به‌همین روش حل کنیم.

در مسئله‌ی انتخاب فعالیت‌ها، ابتدا یک زیر مسئله‌ی کلی را تعریف می‌کنیم و سپس به اثبات دقیق درستی الگوریتم ارائه‌شده می‌پردازیم. یک زیر مسئله، شامل k عدد فعالیت است (در مسئله‌ی اصلی $k = n$). بدون کم شدن از کلیت مسئله می‌توان فرض کرد که شماره‌ی فعالیت‌های یک زیرمسئله به‌همان ترتیب زمان‌های پایان آن‌ها (از کوچک به بزرگ) است. یعنی فرض می‌کنیم که $f_1 \leq f_2 \leq \cdots \leq f_k$.

لم ۷. یک راه‌حل بهینه برای مسئله وجود دارد که شامل کار a_1 است.

اثبات. برهان خلف: فرض A یک راه‌حل بهینه برای مسئله‌ی داده‌شده است که a_1 متعلق به آن نیست. فرض کنید a_j فعالیت با کم‌ترین زمان پایان در A است. بنا به فرض داریم: $f_1 \leq f_j$. حال اگر فعالیت a_j را از A حذف کنیم و به‌جای آن a_1 را قرار دهیم، مجموعه‌ی حاصل نیز یک جواب برای مسئله است، چون a_1 حداکثر می‌تواند با a_j در A اشتراک داشته باشد و با بقیه‌ی فعالیت‌ها اشتراکی ندارد. چون تعداد فعالیت‌های موجود در A و در جواب جدید برابرند، این جواب نیز بهینه است. پس ما یک جواب بهینه‌ی شامل a_1 پیدا کردیم. □

با انتخاب a_1 و سپس حذف کلیه‌ی فعالیت‌هایی که با آن اشتراک دارند، یک زیر مسئله‌ی کوچک‌تر به‌دست می‌آید که آن‌را نیز به همین روش حل می‌کنیم. بنابراین این الگوریتم به یک مرتب‌سازی اولیه نیاز دارد و بقیه‌ی الگوریتم از مرتبه‌ی $O(n)$ انجام می‌شود. پس پیچیدگی الگوریتم $O(n \lg n)$ است.

مثال: الگوریتم پیشنهادی بر روی داده‌های نمونه به‌صورت زیر عمل کرده است.

فعالیت i	زمان شروع (s_i)	زمان پایان (f_i)	انتخاب یا حذف
۱	۱	۴	انتخاب اول
۲	۳	۵	حذف
۳	۰	۶	حذف
۴	۵	۷	انتخاب دوم
۵	۳	۹	حذف
۶	۵	۹	حذف
۷	۶	۱۰	حذف
۸	۸	۱۱	انتخاب سوم
۹	۸	۱۲	حذف
۱۰	۲	۱۳	حذف
۱۱	۱۲	۱۴	انتخاب چهارم

۳.۵.۲ مسئله‌های کوله‌پشتی

دیدیم که مسئله‌های کوله‌پشتی در حالت کلی NP-Complete هستند و برای برخی از آن‌ها راه‌حل پویا با زمان چندجمله‌ای وجود دارد. البته این راه‌حل‌ها چون مقدار حافظه‌ی مصرفی‌شان متناسب است با مقدار اعداد ورودی، راه‌حل واقعاً چندجمله‌ای نیست و به آن شبه چندجمله‌ای (pseudo polynomial) می‌گویند.

برای برخی از مسئله‌های کوله‌پشتی راه‌حل حریصانه وجود دارد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مسئله‌ی خرد کردن پول

من‌خواهیم n تومان را با سکه‌های ۱، ۲ و ۵ تومانی خرد کنیم که مجموع تعداد سکه‌هایی که استفاده می‌کنیم حداقل شود. از هر یک از این سکه‌ها به‌تعداد زیادی در اختیار داریم. این الگوریتم معمولاً برای این مسئله ارایه می‌شود:

> در ابتدا هرچه بتوانیم سکه‌های ۵ تومانی برمی‌داریم تا جایی که مقدار باقی‌مانده از ۵ تومان کم‌تر شود. سپس آن‌قدر سکه‌ی ۲ تومانی برمی‌داریم تا مقدار باقی‌مانده از دو تومان کم‌تر شود و در نهایت مقدار باقی‌مانده را از سکه‌های یک تومانی برمی‌داریم.

این یک الگوریتم حریصانه است، چون در هرمرحله بزرگ‌ترین سکه‌ای که می‌تواند انتخاب می‌کند. آیا این الگوریتم همواره جواب بهینه را می‌دهد؟ جواب این سوال برای این سکه‌ها مثبت است. اثبات این امر به‌تمرین واگذار می‌شود. برای مقادیر دیگری از سکه‌ها (برابر ۱، c، c^2، و ...) هم این الگوریتم درست کار می‌کند. اما اگر به جای سکه‌های ۱، ۲ و ۵ تومانی سکه‌های دیگری در اختیار داشتیم الگوریتم لزوماً درست عمل نمی‌کند. مثلاً اگر به‌جای سکه‌ی ۲ تومانی سکه‌ی ۴ تومانی داشتیم، الگوریتم ۸ تومان را با یک سکه‌ی ۵ تومانی و سه سکه‌ی ۱ تومانی (مجموعا ۴ سکه) خرد می‌کند، درحالتی‌که فقط با دو سکه ۴ تومانی بهتر خرد می‌شود. روشن است که در حالت کلی این مسئله همان مسئله‌ی کوله‌پشتی است که ارزش هر بار ۱ و تعداد هر نوع بار بینهایت است. این مسئله را قبلاً حل کرده‌ایم.

مسئله‌ی کوله‌پشتی با بارهای قابل تقسیم

مسئله‌ی کوله‌پشتی در صورتی‌که بارها را بتوانیم تقسیم کنیم (fractional knapsack problem) نیز راه‌حل ج ازه دارد.

فرض کنید N عدد بار داده شده است که وزن و ارزش بار iام به‌ترتیب برابر W_i و C_i است و می‌خواه ک کوله‌پشتی به‌اندازه‌ی M را با این بارها پر کنیم به‌طوری‌که ارزش بارهای انتخابی بیشترین شود. در این ... مجازهستیم که یک بار را به نسبت دل‌خواه به دو قسمت تقسیم کنیم.

الگوریتم ساده‌ی زیر این کار را انجام می‌دهد:

> بارها را به‌ترتیب ارزش در واحد وزن مرتب می‌کنیم. یعنی فرض می‌کنیم $\frac{c_1}{w_1} \geq \frac{c_2}{w_2} \geq \cdots \geq \frac{c_N}{w_N}$. بارها را به‌همین ترتیب مورد بررسی قرار می‌دهیم. اگر بار i می‌تواند کاملاً در کوله‌پشتی باقی‌مانده قرار بگیرد این کار را می‌کنیم و سراغ بار بعدی می‌رویم. اگر وزن بار iام بیشتر از مقدار باقی‌مانده از کوله‌پشتی است، این بار را به‌نسبت مورد نظر تقسیم می‌کنیم تا کوله‌پشتی کاملاً پر شود. واضح است که این الگوریتم درست کار می‌کند. (اثبات دقیق را به‌خودتان وا می‌گذاریم.)

۴.۵.۴ مسائل زمان‌بندی

در حالت کلی این مسائل NP-Complete هستند اما برای برخی از آن‌ها با روش حریصانه راه حل دقیق دارند.

حالت ساده

یک سرویس‌دهنده و چند کار (job) آماده‌ی گرفتن سرویس داده شده‌اند. زمان مورد نیاز هر کار داده شده است. می خواهیم به تمام این کارها به‌ترتیبی سرویس دهیم و یک پارامتر سیستم را بهینه سازیم. پارامترهای سیستم می‌توانند

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

متوسط زمان پاسخ، حداکثر با متوسط زمان انتظار باشد. مثالی از این مسئله را در صف‌های مختلف در عمل می‌توان دید. چنین صف‌هایی هم در سیستم عامل داریم و بسیاری از الگوریتم‌های زمان‌بندی در این‌جا کاربرد دارند.

به‌بیان دیگر، یک پردازنده، n عدد کار $t_1, t_2, \cdots, t_n$ در زمان صفر داده شده‌اند، به‌طوری‌که زمان مورد نیاز کار t_i برابر s_i است (service time). می خواهیم یک زمان‌بندی یا ترتیب اجرا (scheduling) ارائه دهیم به‌طوری‌که متوسط زمان پاسخ (average response time) حداقل شود. زمان پاسخ کار t_i برابر است با مدت زمان انتظار این کار باضافه‌ی d_i.

می‌توان به‌طور شهودی دید که اگر کارها را به‌ترتیب افزایشی زمان سرویس آن‌ها در صف قرار دهیم، مجموع زمان‌های پاسخ کمینه می‌شود. برای اثبات، فرض می‌کنیم که $d_1 \le d_2 \le \cdots d_n$.

لم ۸. برای مسئله‌ی زمان‌بندی بهینه‌ی یک پردازنده، یک راه‌حل بهینه وجود دارد که در آن اولین کاری که از پردازنده سرویس می‌گیرد، کار t_1 است.

اثبات. اثبات با برهان خلف: فرض کنید که اولین کار t_i باشد ($i > 1$). بنابراین کار t_1 بین کارهای دیگر در صف قرار می‌گیرد. فرض کنید که t_j کاری است که قبل از t_1 انجام می‌شود. نشان می‌دهیم که با تعویض دو کار t_1 و t_j مجموع زمان‌های پاسخ بدتر نمی‌شود.

بدیهی است که تعویض دو کار مجاور تأثیری بر زمان پاسخ کارهای دیگر ندارد. بنابراین فقط مجموع زمان‌های پاسخ کارهای t_1 و t_j مهم است. اگر D زمان اتمام کار جلوی کار t_j باشد، مجموع زمان‌های پاسخ این دو کار در حالت اول برابر است با $A = D + d_j + D + d_j + d_1 = 2D + 2D_j + d_1$ و در حالت دوم برابر است با $B = 2D + 2D_1 + d_j$. بدیهی است که $A - B = d_j - d_1 \ge 0$ (چون d_1 کم‌ترین زمان اجرای کارهاست.)

با این‌ترتیب، روشن است که کار t_1 را می‌توان همواره با کار جلوی‌اش تعویض کرد و هر بار معیار مورد نظر بهتر شود (یا بدتر نشود) تا آن‌جا که این کار اولین کار صف بشود. پس یک راه‌حل بهینه وجود دارد که t_1 اولین کار سرویس گیرنده است. □

با داشتن دو و یا چند پردازنده نیز به همین ترتیب عمل می کنیم. اگر درمسئله‌ی فوق روابط پیش‌نیازی وجود داشته باشد، در حالت کلی مسئله NP-Complete است ولی برای یک یا دو پردازنده راه‌حل چندجمله‌ای وجود دارد.

زمان‌بندی کارها با جریمه‌ی تاخیر

n کار با شماره های ۱ تا n داده شده‌اند که زمان اجرای هر کدام ۱ واحد زمان است. d_i مهلت انجام کار i است و اگر این کار بعد از زمان d_i به‌اتمام برسد جریمه‌ای برابر w_i به آن تعلق می‌گیرد. فرض می‌کنیم که مهلت‌ها اعداد صحیح هستند. هدف تعیین یک زمان‌بندی برای اجرای همه‌ی این کارهاست به‌طوری‌که مجموع جریمه‌ها کمینه شود.

مشاهدات ما از مسئله: فرض می کنیم زمانبندی بهینه را داریم . این زمانبندی را به‌صورت شکل زیر را در نظر میگیریم که در آن هر کار در یکی از بازه‌های زمانی به‌اندازه‌ی ۱ واحد تخصیص داده شده است.

چون به هر کار با تأخیر فقط یک مقدار جریمه تعلق می‌گیرد و میزان تأخیر در این جریمه تأثیر ندارد، روشن است که می‌توان در زمان‌بندی بهینه (یا هر زمان‌بندی داده‌شده) همه‌ی کارهای با تأخیر (late tasks) را در انتهای زمان‌بندی و کارهای بدون تأخیر (early tasks) را در ابتدای آن انجام داد بدون آن‌که در میزان جریمه تغییری داده شود. این کار با تکرار عمل مجاز جابه جایی یک کار بدون تأخیر، مانند t_4 با یک کار با تأخیر که بلافاصله قبل از t_4 اجرا می‌شود انجام داد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

هم‌چنین در زمان‌بندی داده‌شده می‌توان کارهای بدون تأخیر را بر حسب مهلت‌هایشان (از کوچک به بزرگ) مرتب کرد. زیر هر دو کار بدون تأخیر و مجاور t_a و t_b که t_b بلافاصله بعد از t_a اجرا شود، ولی داشته باشیم $d_b < d_a$ را می‌توان با هم جابه‌جا کرد و هر دو کار هنوز قبل از مهلت‌شان اجرا شوند (با توجه به این‌که t_b قبل از زمان d_b اجرا می‌شود و داریم $d_a \geq d_b + 1$، پس با اجرای t_a یک واحد زمانی دیرتر هنوز آن کار قبل از مهلتش اجرا می‌شود. اجرای زودتر t_b مشکلی ایجاد نمی‌کند.)

با این مشاهدات الگوریتم حریصانه‌ی زیر را برای این مسئله پیشنهاد می‌کنیم:

۱. کارها را به‌ترتیب جریمه‌هایشان از بزرگ به کوچک مورد بررسی قرار می‌دهیم. فرض کنید کار t_i را انتخاب کرده‌ایم.

۲. آخرین بازه‌ی زمانی‌ای را پیدا می‌کنیم که t_i را بتوانیم در آن قبل از مهلتش انجام دهیم. برای این کار از بازه‌ی $[d_i - 1, d_i]$ آغاز و بازه‌های سمت چپ آن‌را به‌ترتیب راست به چپ مورد بررسی قرار می‌دهیم. سمت راست‌ترین بازه‌ی خالی محل قرار گرفتن کار t_i است.

۳. اگر بازه‌ی زمانی خالی برای اجرای بدون تأخیر برای کار t_i پیدا نشود، این کار با تأخیر انجام می‌شود و بعد از تعیین همه‌ی کارهای بدون تأخیر زمان‌بندی می‌شود.

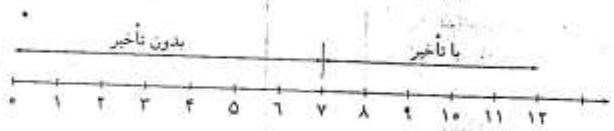
لم ۹. الگوریتم حریصانه‌ی ارائه‌شده یک جواب بهینه برای مسئله‌ی فوق به‌دست می‌آورد.

اثبات. برای اثبات، باید زیر مسئله را به‌درستی تعریف کنیم. فرض می‌کنیم $w_1 \geq w_2 \geq \cdots \geq w_n$. زیرمسئله شامل کارهای t_1 تا t_n است که باید در یک بازه‌ی زمانی mتایی قرار داده‌شوند. با توجه به این‌که قبلاً تکلیف کارهای t_1 تا t_{i-1} مشخص شده است، فرض می‌کنیم که تعدادی (حداکثر برابر $i - 1$ عدد) از بازه‌ها قبلاً پر شده‌اند و قابل استفاده نیستند.

نشان می‌دهیم که اگر امکان اجرای کار t_i قبل از مهلتش باشد، یک راه‌حل بهینه وجود دارد که در آن t_i بدون تأخیر اجرا می‌شود.

فرض کنید که یک راه حل بهینه وجود دارد که در آن t_i با تأخیر اجرا می‌شود. کاری که در بازه‌ی $[d_{i-1}, d_i]$ قرار دارد را t_j بنامید. می‌دانیم که $d_i > d_j$، پس t_j بدون تأخیر اجرا می‌شود. با جابه‌جایی t_i و t_j، در زمان‌بندی جدید t_j با تأخیر اجرا می‌شود که با توجه به این‌که می‌دانیم $w_i \geq w_j$ مجموع جریمه‌ها بدتر نمی‌شود. اگر t_j هم بدون تأخیر اجرا شود، وضعیت بهتر می‌شود.

با تخصیص t_i حداکثر یک بازه‌ی جدید اشغال می‌شود و با کم‌شده این کار، یک زیرمسئله‌ی کوچک‌تر ایجاد می‌شود که به‌همین روش حل می‌شود. □



seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

مثال

کار	۱	۲	۳	۴	۵	۶	۷
d_i	۴	۲	۴	۳	۱	۴	۶
w_i	۷۰	۶۰	۵۰	۴۰	۳۰	۲۰	۱۰

مجموع جریمه‌هایی که تعلق می‌گیرد ۵۰ است.

۵.۵.۴ الگوریتم هافمن

فایلی حاوی n نویسه داده شده است. می‌خواهیم برای هر نویسه کدی طراحی کنیم به‌طوری‌که با استفاده از این کدها (به‌جای کدهای مثلاً ۸ بیتی قبلی) اندازه‌ی فایل جدید (برحسب بیت) کمینه شود. تعداد بیت‌های کدهای طراحی شده می‌تواند متفاوت و دل‌خواه باشد. این بهترین روش برای فشرده‌سازی فایل‌هاست با این شرط که هر نویسه کدگذاری شود. روش‌های دیگری وجود دارند که زیر رشته‌ها را به‌صورتی به کد تبدیل می‌کنند که ممکن است از این الگوریتم بهتر عمل کند.

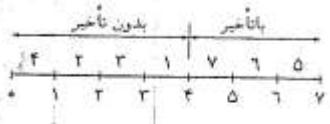
در این مسئله با توجه به این‌که فایل داده‌شده است، بنابراین تعداد تکرار و یا احتمال وقوع هر نویسه را داریم. این الگوریتم را در مواردی مثلاً ارسال اطلاعات ترتیبی و به‌صورت «برخط» (online) بر روی شبکه در صورتی‌که احتمال وقوع نویسه‌ها با تقریب خوبی داشته باشیم، بدون آن‌که کل فایل داده‌شده باشد نیز استفاده می‌شود. توجه کنید که نویسه‌ها لزوماً نباید نویسه‌های مثلاً اسکی باشند.

مسئله به‌صورت انتزاعی از قرار زیر است. n نویسه‌ی a_1 تا a_n و احتمال وقوع f_i برای هر a_i داده شده است ($\sum_i f_i = 1$). می‌خواهیم به‌هر نویسه‌ی a_i کدی به‌طول l_i نسبت دهیم به‌طوری که متوسط طول کدها (یعنی $\sum_{i=1}^{n} f_i l_i$) کمینه شود.

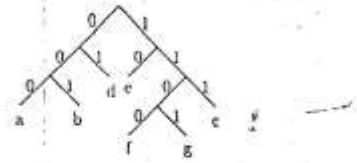
برای آن‌که از فایل دریافت‌شده‌ی کدگذاری شده و نیز جدول کدها بتوانیم فایل اصلی را به‌دست آوریم، روشن است که نباید هیچ کدی زیر رشته‌ی کد دیگر باشد. به‌عبارت دیگر کدها نباید خاصیت پیشوندی داشته‌باشند. با وجود این خاصیت می‌توان کدها را به‌صورت یک درخت دودویی کامل مدل کرد که در آن برگ‌های سمت چپ و راست هر گره به‌ترتیب دارای برچسب‌های ۰ و ۱ هستند و هر نویسه یک برگ این درخت است به‌طوری‌که بیت‌های مسیر از ریشه به آن نویسه کد آن نویسه است. به‌این درخت، درخت هافمن می‌گویند. مثلاً در شکل ۴۴.۴ کدهای a تا g به‌ترتیب برابرند با: ۰۰۰، ۰۰۱، ۱۱۱، ۰۱، ۱۰، ۱۱۰۰، و ۱۱۰۱. توجه کنید که اگر کدی پیشوند کد دیگر باشد، دیگر نمی‌تواند در درخت برگ باشد.

روشن است که متوسط عمق برگ‌های درخت هافمن همان متوسط طول کدهاست.

الگوریتم هافمن با دریافت احتمال وقوع نویسه‌ها، یک درخت هافمن با حداقل متوسط عمق برگ‌ها ایجاد می‌کند. الگوریتم در ابتدا یک گره برای هر نویسه‌ی ورودی ایجاد می‌کند و برای هر گره احتمال وقوع آن نویسه‌را



seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۴.۴: کدگذاری درست را می‌توان با یک درخت دودویی نمایش داد.

نسبت می‌دهد. به‌صورت حریصانه هر بار دو گره‌ی x و y با کم‌ترین احتمال وقوع (f_x و f_y) را پیدا می‌کند، این دو گره را فرزندان یک گره‌ی جدید (به نام مثلاً z) با احتمال وقوع $f_x + f_y$ می‌کند. x و y را حذف و به‌جای آن‌ها z را اضافه می‌کند. الگوریتم این کار را تکرار می‌کند تا این که تنها یک گره با احتمال وقوع ۱ که همان ریشه‌ی درخت است حاصل شود.

پیاده‌سازی الگوریتم به‌صورت زیر است:

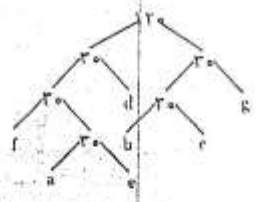
```
Procedure Huffman (C)
Ceate an empty Q
for all c in C do
   Allocate_Node (x); f[x] <- f[c]
   Insert (x, Q)
For i:=1 to n-1 do
begin
   z <-- Allocate_Node();
   x <-- Left[z] <-- Extract_min (Q);
   y <-- Left[z] <-- Extract_min (Q);
   f(z) <-- f(x) + f(y);
   Insert (Q, z)
end;
```

بهترین داده‌ساختار برای Q یک صف اولویت است که اعمال فوق را با $O(\lg n)$ انجام می‌دهد. پس الگوریتم فوق از $O(n \lg n)$ است.

مثال

قابلی به‌اندازه‌ی ۱۲۰ حاوی نویسه‌های زیر است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۵.۲: درخت هافمن برای مثال.

نویسه	فرکانس تکرار
a	۵
b	۱۳
c	۱۲
d	۱۶
e	۴۵
f	۹

اعمال الگوریتم هافمن درخت شکل ۴۵.۲ را ایجاد می‌کند. کدها از روی درخت بدست می‌آید. توجه کنید که کدهای بهینه یکتا نیستند.

[illegible]

[illegible] می‌بینیم که اولین انتخاب درست است. فرض کنید که x و y دو [illegible] یک درخت بهینه وجود [illegible]

[illegible] که [illegible] وجود دارد که مطابق ادعای ما نیست و در آن [illegible] از دست دادن کلیت [illegible] $f(x) <= f(b)$ و [illegible]

[illegible] بدتر نمی‌شود. می‌دانیم [illegible] به‌دلیل مشابه می‌توانیم [illegible] و دیرادر هم و در بیشترین [illegible] می‌توانیم هر دو را با یک نویسه‌ی z با احتمال وقوع [illegible] الگوریتم انجام می‌دهد و یک زیر مسئله با یک نویسه‌ی کم‌تر [illegible] □

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۶.۵.۴ الگوریتم حریصانه‌ی تقریبی برای مسئله‌ی بسته‌بندی

n شیء داده شده‌اند که حجم شیء iام برابر s_i است. s_iها اعداد حقیقی بین صفر و یک هستند. می‌خواهیم این اشیاء را در صندوقچه‌هایی که حجم هرکدام از آن‌ها برابر با ۱ است، بسته‌بندی کنیم به‌طوری‌که تعداد کل صندوقچه‌ها حداقل شود. این مسئله به‌نام بسته‌بندی (bin parking) معروف است. ساده‌ترین الگوریتم حریصانه که برای این مسئله به‌نظر می‌رسد، به این صورت است:

برای قراردادن شیء iام، اگر یکی از صندوقچه‌هایی غیر خالی به اندازه‌ی s_i جای خالی داشت، شیء iام را در آن قرار می‌دهیم و گرنه، آن را در یک صندوقچه‌ی جدید قرار می‌دهیم.

اما متأسفانه این الگوریتم در بسیاری از موارد اشتباه می‌کند. مثلاً اگر حجم اشیاء برابر ۰/۶، ۰/۳، ۰/۳، ۰/۲، ۰/۲، ۰/۲، ۰/۲ باشند، این الگوریتم این اشیاء را در ۳ صندوقچه بسته‌بندی می‌کند: در صندوقچه‌ی اول اشیاء با حجم‌های ۰/۶ و ۰/۳، در صندوقچه‌ی دوم اشیاء با حجم‌های ۰/۳، ۰/۲، ۰/۲، و ۰/۲ و در صندوقچه‌ی سوم یک شیء با حجم ۰/۲ قرار گرفته است. درحالی که می‌توانستیم این کار را با دو صندوقچه نیز انجام دهیم، به این صورت که در صندوقچه‌ی اول اشیاء با حجم‌های ۰/۲، ۰/۲ و ۰/۲ و در صندوقچه دوم اشیاء با حجم‌های ۰/۳، ۰/۳، ۰/۲ و ۰/۲ را قرار دهیم. دیدیم که این الگوریتم ممکن است جواب بهینه را به‌دست ندهد. اما اکنون یک سوال پیش می‌آید: جوابی که این الگوریتم به ما می‌دهد تا چه حد نزدیک به جواب بهینه است؟ قضیه ی زیر تا حدودی به این پرسش پاسخ می‌دهد:

قضیه ۱۲. اگر حداقل تعداد صندوقچه‌های لازم برای بسته‌بندی اشیاء داده‌شده برابر با OPT باشد، الگوریتم فوق این اشیاء را با حداکثر $2 \times OPT$ صندوقچه انجام می‌دهد.

اثبات. در الگوریتم ما ممکن نیست که بیش از یک صندوقچه باقی بماند که کم‌تر از نصف آن پر شده باشد. (چرا؟) بنابراین اگر $S = \sum_i s_i$، الگوریتم ما از حداکثر $\lceil 2S \rceil$ عدد صندوقچه استفاده می‌کند. از طرف دیگر هرداوری که اشیاء را بسته‌بندی کنیم، حداقل به S تا صندوقچه نیاز داریم. بنابراین $OPT >= S$ بنابراین ثابت کردیم که الگوریتم ما حداکثر از $2 \times OPT$ عدد صندوقچه استفاده می‌کند. □ بنابراین الگوریتم حریصانه‌ای که برای این مسئله ارائه دادیم، زیاد هم بد نیست! جوابی که این الگوریتم به ما می‌دهد حداکثر دو برابر مقدار بهینه است. در واقع با استفاده از یک روش پیچیده‌تر می‌توان ثابت کرد که این الگوریتم حداکثر ۱/۷ برابر مقدار بهینه را بدست می‌آورد. البته مثال‌هایی هم می‌توان ساخت که الگوریتم ما دقیقاً ۱/۷ برابر مقدار بهینه را به‌دست آورد. بنابراین با استفاده از این الگوریتم نمی‌توانیم بیشتر از این پیش برویم.

حالا سعی می‌کنیم که کمی الگوریتم فوق را اصلاح کنیم تا نتیجه‌ی بهتری بگیریم. یک کار عاقلانه این است که ابتدا اشیا ی را به‌ترتیب نزولی حجم‌شان مرتب کنیم و سپس همان الگوریتم را در مورد آن‌ها به‌کار ببریم. به‌نظر می‌رسد که با این اصلاح، الگوریتم قدری بهتر شده است، اما هنوز خیلی وقت‌ها اشتباه می‌کند. ثابت شده است که جوابی که این الگوریتم به ما می‌دهد از $OPT + 4$ بیشتر نیست. البته اثبات این موضوع چندان ساده نیست.

البته ذکر این نکته ضروری است که مسئله‌ی بسته‌بندی یک مسئله‌ی NP-Complete است و انتظار ارائه‌ی یک راه‌حل حتی چندجمله‌ای برای آن بیهوده است.

۷.۵.۴ تمرین‌ها

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۱. ثابت کنید که الگوریتم خرد کردن پول اگر سکه های موجود ۱، ۲ و ۵ تومانی باشد، جواب بهینه را به‌دست می‌آورد.

۲. ثابت کنید که اگر سکه های موجود ۱، C، ..... و تومانی باشند الگوریتم خرد کردن پول درست کارمی‌کند.

۳. فرض کنید که سکه هایی با ارزش تومان وجود دارند و از هر کدام از آنها نیز به تعداد نامحدودی در دسترس داریم. می‌خواهیم با استفاده از این سکه ها L تومان پول را خرد کنیم به طوری که تعداد سکه هایی که استفاده می‌کنیم حداقل باشد. را برابر با حداقل تعداد سکه های لازم برای خرد کرد L تومان پول می‌گیریم. ثابت کنید که:
با استفاده از روابط یک الگوریتم برای خرد کردن پول طراحی کنید.

۴. در مسئله کوله‌پشتی، فرض کنید که این شرط را داریم که: ترتیب بسته‌ها وقتی که برحسب وزن‌شان به‌طور صعودی مرتب شوند، همان ترتیبی است که اگر آن‌ها را براحسب ارزش‌شان به‌طور نزولی مرتب کنیم. ثابت کنید که اگر این شرط برقرار باشد، الگوریتم حریصانه درست کار می‌کند.

۵. در مسئله کوله و یک، فرض کنید که n نوع جنس داریم که وزن و ارزش جنس iام به ترتیب برابر و است. از هرکدام از این اجناس نیز فقط یک عدد موجود است. را به این صورت تعریف می‌کنیم: اگر تنها K نوع جنس اول، دوم..... و Kام را در اختیار داشته باشیم، مقدار حداکثر ارزش اجناسی که می‌توان از بین این K جنس انتخاب کرد و در کوله پشتی با ظرفیت W جا داد با است. رابطه ی بازگشتی زیر را برای ثابت کنید و با استفاده از آن الگوریتمی برای حل مسئله کوله پشتی صفر و یک بیابید:

۶. فرض کنید که در یک گراف وزندار کوچکترین زیردرخت فراگیر را بدست آورده‌ایم. حالا می‌خواهیم زیردرخت فراگیری از این گراف را بدست آوریم که پس از کوچکترین زیردرخت فراگیر، از بقیه ی زیردرختهای فراگیر این گراف کم وزن تر باشد. به عبارت دیگر می‌خواهیم زیردرخت فراگیری را بدست آوریم که از نظر کم وزن بودن در مرتبه ی دوم قرار گرفته است. الگوریتم برای حل این مسئله بیابید.

۷. مسئله برنامه ریزی چند پردازنده (Multiprocessor Scheduling) به این صورت مطرح می‌شود: فرض کنید K تا کامپیوتر داریم که سرعت همه‌ی آن‌ها باهم برابر است. می‌خواهیم n تا برنامه را روی این کامپیوترها اجرا کنیم. زمان اجرای برنامه ی iام بر روی هرکدام از این کامپیوترها برابر با t_i است. می‌خواهیم ببینیم که باید هر کدام از برنامه ها را روی کدام کامپیوتر اجرا کنیم که زمان اجرای همه ی برنامه ها، یعنی اولین لحظه‌ای که اجرای تمام برنامه ها به پایان رسیده است، حداقل باشد. یک الگوریتم حریصانه برای این مسئله پیشنهاد می‌کنیم: برنامه ها را به ترتیب نزولی زمان اجرایشان مرتب می‌کنیم. سپس اولین برنامه را به اولین کامپیوتر، دومی را به دومین کامپیوتر،... و Kامین برنامه را به Kامین کامپیوتر اختصاص می‌دهیم. پس از این اولین کامپیوتری که کارش تمام شد، اولین برنامه‌ای که هنوز به هیچ کامپیوتری اختصاص داده نشده است را به آن کامپیوتر می‌دهیم. این کار را تا جایی ادامه می‌دهیم که تمامی برنامه ها بر روی کامپیوترها اجرا شوند. اولا، ثابت کنید که الگوریتم فوق درست نیست ! یعنی مثال نقضی پیدا کنید که الگوریتم فوق نتواند جواب بهینه را برای آن بدست آورد. ثانیا، ثابت کنید که اگر جواب بهینه برای مسئله برابر با T باشد، الگوریتم فوق جوابی برای مسئله بدست می‌آورد که زمان آن از 2T بیشتر نیست.

۸. یک گراف G داده شده است. می‌خواهیم راسهای این گراف را رنگ‌آمیزی کنیم به طوری که شرایط زیر برقرار باشد: الف – هیچ دو راس مجاوری هم رنگ نباشند. ب – تعداد رنگهایی که استفاده می‌کنیم حداقل باشد. ساده ترین الگوریتم حریصانه‌ای که می‌توان برای این مسئله پیشنهاد کرد به این صورت است: ابتدا یک ترتیب دلخواه مانند راس رئوس گراف در نظر می‌گیریم. راس را با رنگ شماره ی ۱ رنگ آمیزی می‌کنیم و پس از آن برای i از ۲ تا n به این صورت جلو می‌رویم: برای رنگ آمیزی راس از اولین رنگی که می‌توانیم استفاده می‌کنیم، یعنی از اولین رنگی که در مجاورت این راس ظاهر نشده است. اگر چنین رنگی موجود نبود، یک رنگ جدید را به مجموعه

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

ی رنگهااضافه می‌کنیم و از آن برای رنگ آمیزی راس استفاده می‌کنیم. به سادگی می‌توانید مثال نقضی برای الگوریتم فوق بیابید. (در واقع هیچ الگوریتم حریصانه‌ای وجود ندارد که جواب بهینه را برای این مسئله پیدا کن- را برابر با تعداد رنگهایی می‌گیریم که الگوریتم فوق برای رنگ آمیزی گراف از آن استفاده می‌کند، اگر ترتیب اولیه ی O را برای رئوس انتخاب کرده باشیم. همچنین C (G) را برابر با حداقل تعداد رنگهایی که برای رنگ آمیزی گراف G لازم است می‌گیریم ثابت کنید که: الف – برای هر گراف G، ترتیب O وجود دارد که شود. به عبارت دیگر در هر گراف اگر ترتیب اولیه O، ترتیب مناسبی باشد، الگوریتم درست عمل می‌کند. ب – برای هر عددٌ حقیقی ، گراف G و ترتیب O برای رئوس آن وجود دارد که نسبت از بیشتر شود. به عبارت دیگر الگوریتم حریصانه می‌تواند به مقدار دلخواهی اشتباه کند.

۹. گراف بدون جهت G داده شده است. مسئله پوشش راسی (Vertex-Cover) به این صورت مطرح می‌شود: کوچکترین زیرمجموعه ی C از مجموعه ی رئوس گراف را پیدا کنید که برای هر یک از یالهای گراف، حداقل یکی از دو سر این یال در مجموعه ی C باشد. یک الگوریتم مکاشفه‌ای ساده برای پیدا کردن کوچکترین پوشش راسی به این صورت است: ۱) C را مساوی مجموعه ی تهی و E' را مساوی مجموعه یالهای گراف قرار بده. ۲) یال دلخواه uv از مجموعه ی E' را انتخاب کن ۳) دو راس u و v را به مجموعه ی C اضافه کن ۴) تمامی یالهایی که لااقل یکی از دو سرشان u یا v باشد را از مجموعه ی E' حذف کن. ۵) اگر E' مجموعه تهی نیست، به مرحله ی ۲ برگرد. ۶) مجموعه ی C را به عنوان خروجی الگوریتم برگردان.

ابتدا مثال نقضی پیدا کنید که الگوریتم فوق درست کار نکند و سپس ثابت کنید که برای هر گراف G جوابی که الگوریتم فوق می‌دهد حداکثر دوبرابر مقدار بهینه عضو دارد.

۱۰. راننده‌ای می‌خواهد با ماشینش از یک شهر به شهر دیگری سفر کند. ماشین با یک بنزین پر می‌تواند n کیلومتر حرکت کند. در مسیر مسافرت این راننده k تا پمپ بنزین وجود دارد. (پمپ بنزین اول در شهر مبدأ و پمپ بنزین kام در شهر مقصد است) فاصله‌ی پمپ بنزین iام با پمپ بنزین i + ۱ام برابر با d_i کیلومتر است. اگر اعداد n و d_1 و ... و d_k داده شده باشند، می‌خواهیم حداقل تعداد پمپ بنزین هایی را پیدا کنیم که راننده می‌تواند با پرکردن باک بنزین ماشینش در این پمپ بنزینها مسافرت خود را انجام دهد. یک الگوریتم حریصانه برای حل این مسئله پیشنهاد کنید و آن را ثابت کنید.

۱۱. n نقطه با طول‌های x_1 و ... و x_n روی محور طول‌ها داده شده‌اند. الگوریتمی طراحی کنید که کمترین تعداد بازه‌های به‌طول ۱ را پیدا کند که تمامی این نقاط را بپوشانند. درستی الگوریتم خود را ثابت کنید.

۱۲. در یک فروشگاه n مشتری منتظر هستند تا کارشان انجام شود. کار مشتری شماره ی i به‌اندازه‌ی t_i دقیقه طول می‌کشد. یک نفر کارمند متصدی انجام کار این مشتری‌هاست و در هر لحظه می‌تواند کار فقط یک مشتری را انجام دهد. می‌خواهیم ببینیم که کارمند این فروشگاه باید کار این مشتری‌ها را به چه ترتیبی انجام دهد تا مجموع زمان معطل شدن این مشتری ها کمینه شود. (زمان معطل شدن یک مشتری برابر با زمانی است که انجام کار او به اتمام می‌رسد.) یک الگوریتم حریصانه برای حل این مسئله پیدا کنید و سپس درستی الگوریتم خود را اثبات نمایید.

۱۳. n برنامه با طول‌های l_1، l_2، dots، و l_m باید روی یک نوار مغناطیسی ذخیره شوند. می‌دانیم که احتمال این‌که بخواهیم برنامه ی شمارهٔ ی i را از روی نوار بخوانیم برابر با p_i است. اگر برنامه‌ها روی نوار مغناطیسی به ترتیب شماره‌هایشان ذخیره شده باشند (یعنی برنامه‌ی اول در ابتدای نوار، برنامه‌ی دوم پس از آن، و ...) مقدار زمانی که طول می‌کشد تا بتوانیم برنامه‌ی شماره‌ی i را از روی نوار بخوانیم، متناسب با $\sum_{k=1}^{i} l_k$ است. بنابراین زمان خواندن یک برنامه از روی نوار به طور متوسط متناسب با $T = \sum_{i=1}^{n} p_i \sum_{k=1}^{i} l_k$. هدف این است که ترتیب برنامه‌ها روی نوار مغناطیسی را طوری تعیین کنیم که T حداقل شود.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

الف – با ارائه‌ی یک مثال ثابت کنید که اگر برنامه‌ها را به‌ترتیب صعودی طول‌شان روی نوار ذخیره کنیم، ترتیب ذخیره‌شدن برنامه‌ها لزوماً بهینه نیست.

ب – با ارائه‌ی یک مثال ثابت کنید که اگر برنامه‌ها به ترتیب نزولی طول‌شان روی نوار ذخیره شوند، ترتیب ذخیره شدن برنامه‌ها لزوماً بهینه نیست.

ج – ثابت کنید که اگر برنامه‌ها را به‌ترتیب نزولی مقدار $\frac{l_i}{p_i}$ بر روی نوار ذخیره کنیم، مقدار $\overline{T}$ مینیمم می‌شود، یعنی این ترتیب، بهینه است.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

۶.۴ روش‌های جست‌وجو

در بخش‌های قبل روش‌های تقسیم‌وحل، پویا، و حریصانه که عمدتاً برای به دست آوردن راه‌حل‌های سریع و چندجمله‌ای برای مسایل استفاده می‌شوند، مورد بررسی قرار گرفت. ولی پیدا کردن چنین الگوریتم‌هایی کار ساده‌ای نیست و همه‌ی مسایل را نمی‌توان با روش‌های مذکور حل کرد.

اگر مسئله‌ای با هیچ یک از روش‌های فوق، قابل حل نبود، ممکن است تنها راه‌حل آن جست‌وجوی فضای حالت باشد؛ یعنی برای پیدا کردن جواب، کلیه‌ی حالت‌های مسئله را مورد بررسی قرار می‌دهیم. این جست‌وجو معمولاً به یکی از روش‌های زیر انجام می‌گیرد:

۱.۶.۴ روش پس‌گرد

پس‌گرد[۲۶] روشی است که کلیه‌ی فضای حالت قابل قبول را با نظم خاصی ایجاد و جست‌وجو می‌کند. به این صورت که در مرحله‌هایی که با چندین انتخاب روبرو می‌شود، یکی از انتخاب‌ها را با فرض درست بودن دنبال می‌کند، اگر به جواب نرسید، با پس‌گرد (backtrack) انتخاب خود را عوض می‌کند و این کار را تا تمام شدن انتخاب‌ها ادامه می‌دهد. این روش می‌تواند با یافتن اولین جواب متوقف شود و یا این که جست‌وجو را برای یافتن کلیه‌ی جواب‌ها ادامه دهد.

الگوریتم‌های پس‌گرد معمولاً دارای هزینه‌ی نمایی می‌باشند.

مسئله‌ی هشت‌وزیر

```
for i1 := 1 to 8 do
  for i2:= 1 to 8 do
    for i3:= 1 to 8 do
      ....
        for i8:= 1 to 8 do
          try <- (i1,i2,...,i8)
          if solution (try) then write(try)
```

بردار k-promising ...

[۲۶] backtracking

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

```
procedure N-Queens (k, col, diag45, diag135)
(try [1..k] is k-promising
 col= {try[i]: 1<= i <=k}
 diag45 = {try[i]+i+1: 1<= i <=k}
 diag135 = {try[i]+i-1: 1<= i <=k}
)
if k = N  (an N-promising verstor is a solution)
   then write try
else { find (k+1)-promising extensioun)
   for j <- 1 to N do
      if (j not in col) and (j+k not in diag45)
                        and (j+k not in diag135)
      then try[k+1] <- j
      N-Queens(k+1, col+{j}, diag45+{j-k}, diag135+{j+k})
```

برای $n = 13$، ۴۷۹۰۰۱۶۰۰ جای‌گشت وجود دارد. اولین جواب در ۴۵۴۶۰۴۴امین حلقه به‌دست می‌آید، ولی درخت حالت در روش پس‌گرد ۸۵۶۱۸۹ گره دارد که در ۲۶۲امین گره یک جواب به‌دست می‌آید.

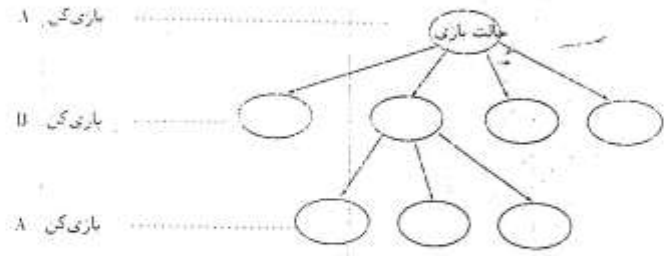
۲.۶.۴ درخت بازی

یکی از روش‌های جست‌وجوی فضای مسئله، استفاده از درخت بازی[۲۷] می‌باشد که جهت تعیین استراتژی برد و انجام بهترین بازی ممکن در هر مرحله به‌کار می‌رود. هر گره درخت بر یک وضعیت از بازی دلالت می‌کند و وضعیت‌هایی که با حرکت بعد قابل تولید هستند، به عنوان فرزندان این گره در نظر گرفته می‌شوند (مانند شکل ۱.۴۶.۴).

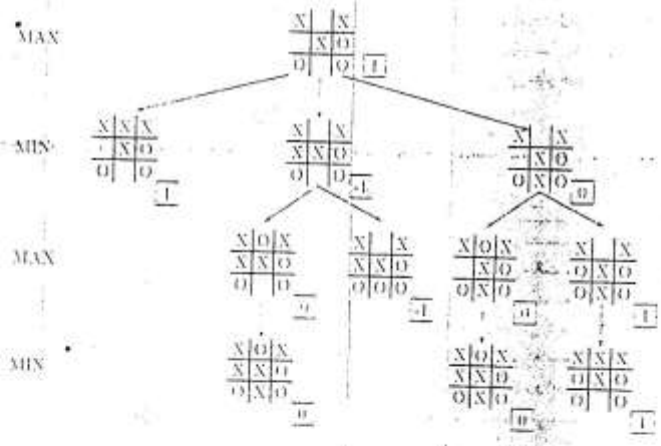
برگ‌ها خاتمه‌ی بازی را نشان می‌دهند و هر مسیر تا رسیدن به یک برگ، بیان‌گر یک بازی است که یک یال این مسیر توسط بازیکن اول انتخاب شده و یال بعد را بازیکن دوم انتخاب کرده است. به هر گره عددی نسبت داده می‌شود که بستگی به وضعیت بازی دارد. برای روشن شدن مطلب به مثال زیر توجه کنید:

در بازی X-O از درخت بازی می‌توان به صورت نشان داده شده در شکل ۴۷.۴ در تصمیم‌گیری برای رسیدن به هدف نهایی که قرار دادن حروف مشابه به صورت دنبال هم می‌باشد، استفاده نمود. در حالت نهایی (برگ‌ها) اگر X برنده باشد عدد ۱، اگر O (حریف) برنده باشد عدد ۱- و در غیر این صورت عدد صفر به آن برگ نسبت داده می‌شود. در سطح پدر اگر حرکت (انتخاب یال) از X باشد بیشینه‌ی عدد فرزندان به این گره نسبت داده می‌شود و اگر حرکت از O باشد، کمینه‌ی آنها. بدین معنی که ما وضعیتی را انتخاب می‌کنیم که بیشترین امتیاز را دارد و حریف هم بهترین بازی خود را ارایه می‌دهد تا کم‌ترین امتیاز نصیب ما شود. از این رو این درخت را min-max سر می‌گویند.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۶.۴: درخت بازی.



شکل ۴۷.۴: درخت بازی در بازی X-O

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

پیاده‌سازی درخت بازی

```
function Search(B: BoardType; Mode: ModeType): real;
var
  C: BoardType;
  Value: real;
begin
  if B is a leaf then
    return(Payoff(B))
  else
  begin
    if Mode = Max then
      Value := -1
    else
      Value := +1;
    for each child C of board B do
      if Mode = Max then
        Value := Max(Value, Search(C, Min))
      else
        Value := Min(Value, Search(C, Max));
    return(Value);
  end;
end;
```

مسئله‌ی فوق تمام فضای حالت را جست‌وجو می‌کند. اما در بعضی موارد به دلیل زیاد بودن حالت‌ها چنین امری امکان‌پذیر نیست. مثلا در بازی شطرنج ساخت درخت تا رسیدن به حالت‌های نهایی که بتوان به آن امتیاز نسبت داد مقدور نیست. لذا در چند سطح که امکان دارد، ساخته می‌شود و در سطح آخر با توجه به وضعیت مهره‌ها در صفحه، عددی را حدس زده، و به آن نسبت می‌دهیم. هرقدر که این حدس قوی‌تر و تعداد سطوح ایجاد شده‌ی درخت بیش‌تر باشد، به ضمن میزان بازی به‌تری انجام می‌گیرد.

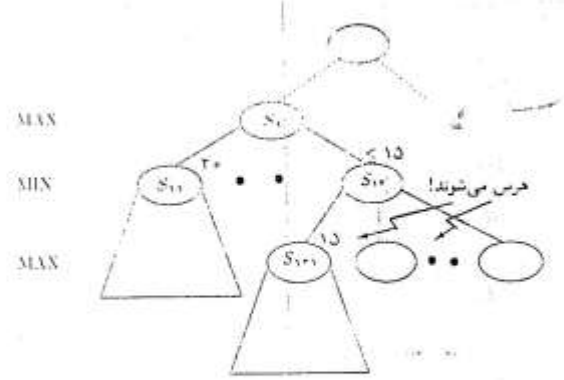
۳.۶.۴ محدود کردن فضای جست‌وجو

در مسائل فوق سعی بر این بود که تمامی فضای حالت مورد جست‌وجو قرار گیرد. اما با استفاده از روش‌های زیر می‌توان فضای جست‌وجو را کاهش داد:

۱. هرس کردن[۲۸]

۲. انشعاب و تحدید[۲۹]

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۸.۴: با بازگردن S_{101} بی‌تاثیر بودن S_{10} مشخص می‌گردد و بقیه‌ی فرزندان آن هرس می‌شوند.

هرس کردن

گاهی اوقات جست‌وجوی برخی از شاخه‌ها بی‌تاثیر است و می‌توان از جست‌وجوی آن صرف نظر کرد. در شکل ۴۸.۴ چون S_{11} امتیازی معادل ۲۰ دارد، درنتیجه امتیاز S1 بیشتر از ۲۰ خواهد بود و در شاخه‌ی S_{10} پس از باز کردن S_{101} با امتیاز ۱۵ متوجه می‌شویم که امتیاز S_{10} کمتر از ۱۵ خواهد بود. و از آن‌جا که امتیاز S_0 بیش‌تر از ۲۰ است لذا از باز کردن بقیه‌ی فرزندان S_{100} صرف نظر می‌کنیم و آن‌ها را با تمامی زیرشاخه‌هایشان هرس می‌کنیم.

در مورد درخت بازی این روش را هرس [illegible] گویند. شکل ۴۹.۴ نمونه‌ای [illegible] را بررسی می‌کند.

انشعاب و تحدید

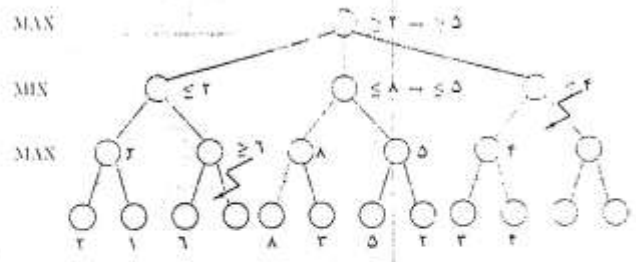
در این روش نیز مانند هرس کردن، از بازکردن برخی از شاخه‌ها خودداری می‌شود. به این صورت که با تجزیه و تحلیل اطلاعات محلی موجود در بعضی از مرحله‌ها، مشخص می‌گردد که از این شاخه نسخه‌ی مطلوبی حاصل نخواهد شد. لذا از باز کردن آن صرف نظر می‌شود. هم‌چنین با توجه به همین اطلاعات، تخمین می‌گیریم که آیا گره‌ای از شاخه‌های درخت را مورد جست‌وجو قرار دهیم. برای روشن شدن مطلب به مثال زیر توجه کنید.

فروشنده‌ی دوره‌گرد

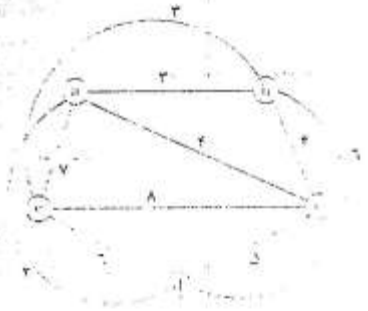
فرض کنید شهرها و راه‌های بین آن‌ها مطابق شکل ۵.۴ داده شده است. هدف پیدا کردن دوری با حداقل وزن ممکن است، که از کلیه‌ی رأس‌ها عبور کند و از هر رأس فقط یک بار بگذرد.

حالت مسئله: تعدادی از یال‌ها انتخاب شده‌اند. پس قطعاً این یال‌ها در مسیر سفر وجود دارند. هم‌چنین تعدادی از یال‌ها حذف شده‌اند، که نمی‌توان از آن‌ها در مسیر سفر استفاده کرد. مثلاً کلیه‌ی مسیرهایی که یال‌های (۱، ۲) و (۱، ۳) را

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۴۹.۴: نمونه‌ای از یک α-β pruning



شکل ۵۰.۴: مسیر شهرها در مسئله‌ی فروشنده‌ی دوره‌گرد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com

شامل می‌شود و بال iii در آن مسیرها وجود ندارد. یک حالت از مسئله می‌باشد. این حالت را به صورت زیر نمایش می‌دهیم:

(i, ii, iii)

تعیین حد: در هر حالت برای هر رأس دو بال با کمترین وزن، از «بال‌های موجود» انتخاب می‌کنیم. بدیهی است که هیچ مسیر هامیلتونی پیدا نمی‌شود که وزن آن کم‌تر از نصف مجموع وزن‌های این بال‌ها باشد. این حد همان اطلاعات محلی در حالت محسوب می‌شود. مثلاً در حالتی که هیچ بالی حذف نشده است، حد به طریق زیر محاسبه می‌شود:

$$\begin{array}{ll} a: & ۲ + ۳ \\ b: & ۳ + ۳ \\ c: & ۴ + ۴ \\ d: & ۲ + ۵ \\ e: & ۳ + ۶ \\ \hline & ۳۵ \div ۲ = ۱۷.۵ \end{array}$$

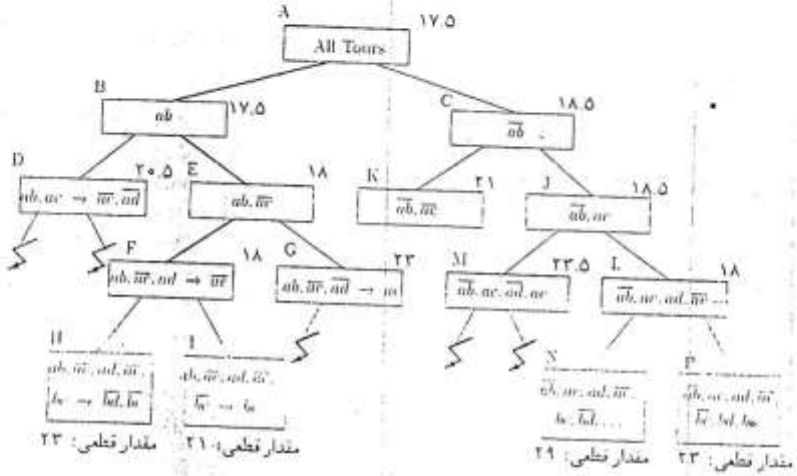
لذا هیچ مسیری با وزن کمتر از ۱۷/۵ و در واقع کم‌تر از ۱۸ نداریم (چون وزن‌ها عدد صحیح هستند.)

تصمیم‌گیری برای باز کردن یا بازنکردن شاخه‌ها با توجه حد آن صورت می‌گیرد. هم‌چنین در انتخاب یک شاخه برای دنبال کردن، اولویت را به شاخه‌ای می‌دهیم که حد کمتری دارد. برای مثال در شکل ۶.۴ ابتدا مسیری که پررنگ‌تر رسم شده، جست‌وجو می‌شود.

در شکل فوق پس از رسیدن به I با ارزش ۲۱ و بازگشت به مرحله‌ی E، مرحله‌ی i هرس می‌شود. زیرا مسیرهایی که از G تولید می‌شوند، بزرگتر یا مساوی ۲۳ هستند. و هم‌چنین در مرحله‌ی D مسیرهایی که تولید می‌شوند، بزرگ‌تر یا مساوی ۲۱ هستند لذا D هم هرس می‌شود. در یک سطح بالاتر چون ۲۱ از ۱۹ بزرگتر است، لذا اعمال فوق را دوباره تکرار می‌کنیم و اتفاقاً با توجه به شکل مسیری به طول ۱۹ بدست می‌آید که جواب مسئله است در حالی که D,G,M,K باز نشده‌اند.

نکته‌ی مهم در حل این گونه مسائل به دست آوردن یک حد (lb=lower bound) صحیح می‌باشد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com



شکل ۵۱.۴: درخت جست‌وجوی مسئله‌ی فروشنده‌ی دوره‌گرد.

seyed hamed hoseinian
Software Engineering pnub
saeeditpnu.blogfa.com